Investigation of Optimization Techniques for Scheduling Precedence Computations with Communication Costs

by

Homam Marwan Rashad Najjari

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

June, 1996

Investigation of Optimization Techniques for Scheduling Precedence Computations with Communication Costs

BY

Homam Marwan Rashad Najjari

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

June 1996

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by

Homam Marwan Rashad Najjari

under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Thesis Committee:

Dr. Hussein Al – Muallim
Chairman

Dr. Mayez Al – Mouhamed
Co-Chairman

Dr. Muslim Bozyigit
Member

Dr. Subbarao Ghanta
Member

Department Chairman

Dean, College of Graduate Studies

3.7.96
Date

To my father and mother

whose support and prayers

led to this achievement.

To my dear brothers

Rashad and Maen.

إهداء

الحمد لله رب العالمين الذي علم بالقلم ، علم الانسان ما لم يعلم ، والشكر له على توفيقه وامتنانه ، والصلاة والسلام على سيدنا محمد المبعوث إلى الناس كافةً من عرب وعجم ، داعياً إلى الله بإذنه ونور الهُدى ، معلم الإنسانية ومرشدها إلى النور بإذن ربها ، فعليه صاحب الحوض المورود ، والمقام المحمود ، خاتم النبيين والمرسلين أفضل صلاة وأزكى تسليم ، وعلى آله وصحبه أجمعين .

أما بعد : فهذا عملي المتواضع ، وعصارة فكري ، وباكورة إنتاجي العلمي ، أهديه وما بذلته من جهد مضني وأمضيت فيه من سويعات عمري وشبابي ، ثمرة يانعة بسعادة غامرة ، وقلب ملئه الحب والوفاء ووجدان عامر بالعرفان ، أهديه إلى من كان السبب بعد الله في وجودي ، إلى من تعهد الغرس حتى أثمر ، ومن حبب إليَّ طلب العلم وزينه في قلبي ، فرغبني فيه ، وحضَّني عليه . وأهديه كذلك إلى معلميَّ الخير والباذلين فيه كل غال وثمين ، أينما كانوا وحيثما وجدوا ، وإلى كل الذين منحوني النصح ومدوا إليَّ أيديهم البيضاء وتمنوا لي الفلاح .

إليكم جميعًا أيها السادة الكرام ، أتقدم متواضعًا بهذا الاسهام ، راجيًا لكم على الدوام العافية وحسن المثوبة في الدنيا والدار الآخرة . وكما بدأت هذا الإهداء أختمه بأن الحمد لله رب العالمين والصلاة والسلام على رسوله الصادق الأمين .

والسلام عليكم ورحمة الله وبركاته

هُمام مروان رشاد نجّاري

Acknowledgments

All praise be to Allah for his limitless help and guidance. Peace and blessings of Allah be upon his prophet Muhammad.

Acknowledgment is due to King Fahd University of Petroleum and Minerals for the generous help and support for this research.

I would like to thank my thesis chairman, Dr. Hussein Al-muallim, Assistance Professor of Information and Computer Science, for his remarkable suggestions and guidance. I would also like to express my profound gratitude and appreciation to my thesis co-chairman, Dr. Mayez Al-Mouhamed, Associate Professor of Computer Engineering, with whom most of this work was done. Also I would like to thank Dr. Muslim Bozyigit, Associate Professor of Information and Computer Science, and Dr. Subbarao Ghanta, Associate Professor of Information and Computer Science, for their consistent support and patience through out the course of this thesis

I also wish to thank the faculty, graduate students, and the staff members of the Information and Computer Science Department. I would also like to thank Dr. Mousa Rayhan for his helpful comments in writing the arabic dedication. Special thanks go to my cousin and her husband Dr. Jamal Enebtawi for their continuous help and support. The encouragement and good wishes of my friends, specially my roommate Tareq Al-Naffouri, are also worthy of acknowledgment. Finally, I would like to dedicate this thesis to the rosy part of my life, to my parents and dear brothers for their continuous prayers, encouragement and moral support.

Contents

List of Tables

List of Figures

Abstract

Name: Homam Marwan Rashad Najjari

Title: Investigation of Optimization Techniques for Scheduling Precedence Computations with Communication Costs

Major Field: Computer Science

Date of Degree: June, 1996

Efficient scheduling of precedence computations with communication is crucial for distributed systems. A precedence computations with communication is modeled as a directed acyclic graph. The objective here is to find a more refined strategy based on the Iterative Refinement Scheduling concept, that was previously developed, to approximate the task level. This work proposes a global scheduling heuristic that combines the refined estimate of task levels with management of processor idle times. Extensive testing of the proposed heuristic is conducted by altering the granularity, parallelism, and system topology. Analysis showed that, at coarse-grain computations, better performance can be achieved by reducing processor idle times. For fine-grain computations, however, better performance requires higher precision selection of critical tasks. Testing proved that the proposed heuristic outperforms other recently reported heuristics, and it generates near-optimum solutions. The time complexity of the proposed heuristic is $O(pn^2)$, where p and n are the numbers of processors and tasks, respectively.

Master of Science Degree

King Fahd University of Petroleum and Minerals
Dhahran - Saudi Arabia

June, 1996

بسم الله الرحمن الرحيم

خلاصـة الرسالـة

اسم الطالـــب : هُمام مروان رشاد نجّاري

عنوان الرسالة : البحث في التقنيات التفضيلية لجدولة العمليات الحسابية ذات الأولوية والاتصالات

التخصـــص : علوم الحاسب الآلي

تاريخ الشهادة : حزيران / يونيو ١٩٩٦م

تعتبر الجدولة الفعالة للعمليات الحسابية ذات الأولوية والاتصالات ضرورة للنظم الموزعة . تتمثل العمليات الحسابية ذات الأولوية والاتصالات بمخطط بياني موجه غير حلقي . الهدف هنا هو إيجاد استراتيجية أفضل لتقريب مستوى المهمات بالاعتماد على فكرة الجدولة التكرارية التحسينية التي تم تطويرها مسبقًا . يقترح هذا البحث أسلوب جدولة على أساس الأفضلية العامة بحيث يضم الاستراتيجية المقترحة لتقريب مستوى المهمات ، مع إدارة أفضل لفترات عَطَلَة المعالِجات . وقد تم القيام باختبارات مكثفة لتقويم أداء الأسلوب المقترح ، بتغيير في مستوى الحبيبية للحسابات ، ودرجة التوازي ، وطبيعة شبكة المعالِجات . أظهرت الدراسة أهمية تقليل فترات عَطَلَة المعالِجات عند جدولة الحسابات ذات المستوى الحبيبي الخشن . أما بالنسبة لجدولة الحسابات ذات المستوى الحبيبي الناعم ، فإن تحسين الأداء يتطلب دقه أكثر في اختيار المهمات الحرجة . أثبتت التجارب أن أداء الأسلوب المقترح يفوق أداء الأساليب الموجودة حاليًا . بالإضافة إلى ذلك ، يتطلب هذا الأسلوب عددًا منخفضًا ومقبولاً من الخطوات الحسابية .

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران - المملكة العربية السعودية

حزيران / يونيو ١٩٩٦م

Chapter 1

Introduction

Medical diagnosis, weather forecasting, genetic engineering, predictive modeling simulations, and interactive graphics are some examples of scientific and engineering applications where high performance computers are of great demand. High performance is not only due to the use of faster and more reliable hardwares, but also to major improvement in computer architectures and processing techniques [34] [23].

It is crucial for parallel processing systems to have efficient operating systems. This is because the throughput of the whole parallel system is generally affected by how efficient its operating system maps a given set of computation tasks to the set of available processors in the parallel system. Given a set of computation tasks along with their computation times, precedence constrains, inter-task communication costs, and a multiprocessor system, the scheduling problem can be defined as mapping the given set of tasks to the available processors so that the resulting

schedule length is minimized while the precedence constraints are preserved.

The set of computation tasks along with their specifications can be modeled as a directed acyclic graph (DAG). The critical path (CP) of a given computation graph is defined as the largest sum of computations and communication costs along a path from an entry node to an exit node. It is crucial to schedule tasks along the critical path as early as possible. In order to minimize the resulting schedule length by not delaying tasks that lie on the CP. However, finding the critical path of a given computation graph is very dependent on the task-processor mappings, which are not known before hand.

In fact, finding the optimum schedule of a given computation graph is very expensive and was proved to be an NP-complete problem. So various heuristics and means have been proposed in the literature to overcome this problem. Branch and bound, simulated annealing, and genetic algorithms explore the search space until they find some acceptable schedule. These techniques are classified as search-based ones and are very expensive in terms of their execution times. Task Duplication (TD), and Task Clustering (TC) are some other scheduling heuristics that overcome this problem partially, but they have their own deficiencies. TD suffers from increasing the space complexity, and the management overhead of duplicated tasks over different processors. Further, TC algorithms assumes unbounded number of processors. In fact their resulting schedules need to be mapped to the available processors in the system, which is another very expensive problem.

Priority-based scheduling is another scheduling approach, which utilizes a selection strategy to pick a task among a set of ready to run tasks and assign it to an idle processor in an attempt to minimize the overall schedule finish time of a given computation graph. We can distinguish between two classes of selection strategies. These are local selection strategies and global selection strategies . Local strategies lack the utilization of global information. This is in contrast to global strategies where the notion of task level is fundamental.

The task level value defines how much computations and communication costs are left from the start of the task up to an exit node. Task level values can be used to discriminate critical tasks that have high task level values, from less critical tasks that have lower task level values.

Iterative Refinement Scheduling (IRS) is a scheduling optimization technique used to enhance the approximate evaluation of task level values computed by a scheduling heuristic. Basically, IRS utilizes some global scheduling heuristic to schedule the given computation n times. It passes the approximate evaluation of task level values from one scheduling pass to the next scheduling pass to use them in the evaluation of task priorities. Though IRS is able to improve the quality of the resulting schedules, its behavior has not been well explored yet.

The remaining chapters of this thesis will concentrate on scheduling computation tasks with communication times on loosely coupled multiprocessor systems. This thesis presents a new scheduling heuristic for scheduling precedence computation

graphs with communication costs on a general loosely coupled parallel system. Extensive empirical testing of this heuristic along with the best reported scheduling heuristics has been performed. The thesis is organized as follows. Chapter 2 presents the system model definition along with the common terms and the nature of the scheduling problem. In chapter 3 we present our review of bounds on scheduling and the most relevant scheduling approaches. The design and analysis of the new scheduling heuristic is introduced in chapter 4. Chapter 5 summarizes the empirical testing and the performance evaluation of the new heuristic. This chapter also includes the comparison with other reported scheduling heuristic. Chapter 6 presents the conclusion about this M.S. thesis.

1.1 Parallel Processing

The bandwidth of a single CPU has been known to be technologically limited. The natural consequence of this has been to explore increasing the number of processing units that will work simultaneously on a given set of computation tasks. This new processing paradigm is called parallel processing. Hwang [19] defined this concept as follows: " Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process".

In fact, parallel processing is in contrast to sequential processing, as it emphasizes the exploitation of concurrent events in the computation process. Parallel

processing can be conducted at four distinct programmatic levels [19]. The highest level of parallel processing is to explore concurrency among multiple programs through multi-programming, time-sharing, and multi-processing. The next level is conducted at the procedure or task level within the same program. The third level of parallel processing is performed at the instruction level. The last level, is the intrainstruction level where concurrency is exploited among operations within each instruction. Usually the highest level of parallel processing is implemented as part of the operating system. In fact, it is clear that the hardware role increases in the direction from high level towards low level, and the opposite applies for the software role.

1.2 Architectural Configurations of Parallel Systems

Parallel computers or systems are those systems that incorporate parallel processing and they can be grouped in three architectural configurations [19]. The first architecture is called a *pipeline computer* in which overlapped computations are performed to exploit temporal parallelism. The second architecture is called an *array processor.* It uses synchronous multiple arithmetic and logic units called processing units PE's, that can operate in parallel. These are passive devices without decoding abilities, and all of them are synchronized to do the same operation. The last one is the *mul-*

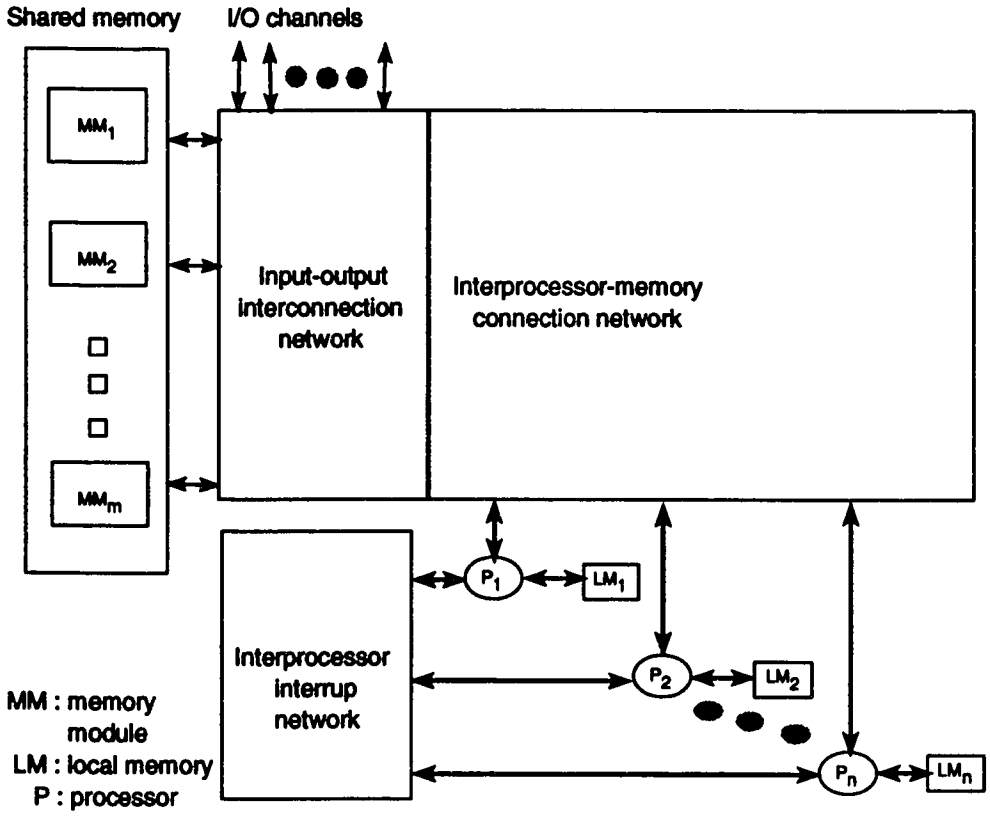

Figure 1.1: A functional design of a multiprocessor system.

tiprocessor system . This system consists of two or more asynchronous comparable processors that can share access to a common set of memory modules, I/O channels and peripheral devices. Inter-processor communication is conducted through shared memory, or an interrupt controller (see figure 1.1).

Flynn [13] has classified digital computer systems into four groups according to the multiplicity of instruction and data streams. They are listed below in increasing multiplicity of hardware.

- Single instruction stream-single data stream ($SISD$)
- Single instruction stream-multiple data stream ($SIMD$)
- Multiple instruction stream-single data stream ($MISD$)
- Multiple instruction stream-multiple data stream ($MIMD$)

The *SISD* group represents the majority of serial computers available today. In fact, most $SISD$ uniprocessor systems are pipelined. A $SISD$ system may have multiple processing units, but all of them are under the supervision of one main control unit.

The *SIMD* group corresponds to array processor based systems, where all the processing units ($PE's$) execute the same instruction broadcasted form the control unit , but on different data sets from distinct data streams.

No real computers of the *MISD* class exists commercially, where a group of n processors receive n different instructions operating on the same data or data derived from it. The output of one processor is used as an input for the next processor.

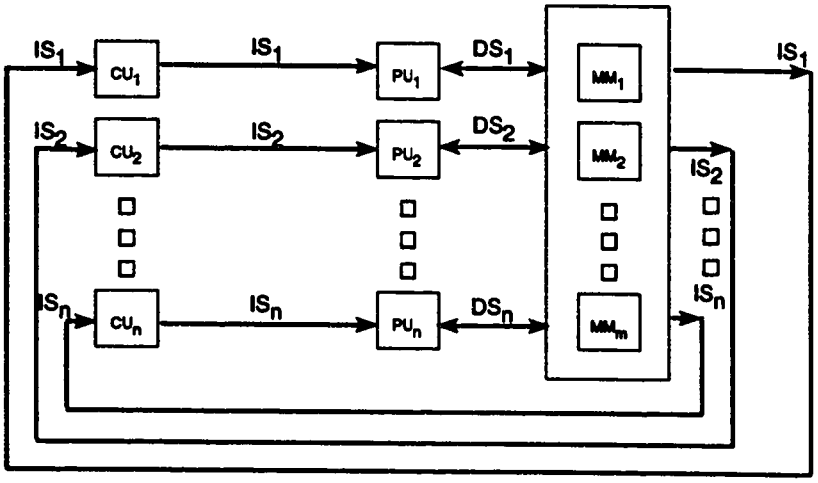

Figure 1.2: An MIMD computer

Most multiprocessor systems can be classified in the *MIMD* class (see figure 1.2). In intrinsic $MIMD$ model, the n data streams of the n processors are derived form the same data space shared by all of them. When the n data streams are derived from disjoint data spaces in the shared memory modules, then this is called multiple $SISD$ and denoted by $MSISD$. Intrinsic $MIMD$ computers with high inter-processor interactions are referred to as *tightly coupled*; otherwise, they are considered *loosely coupled*. In a tightly coupled system, like array processor based systems, or multiprocessor systems, the memory bandwidth limits the number of processors that can be usefully deployed. This problem is reduced by associating a private memory with each processor

1.3 Performance of Parallel Computers

One might expect that a parallel computer with n identical processors working concurrently on a single problem would be n times faster than a single processor computer. In real life applications, however, the speedup is much less because of the fact that some processors are idle at a given instance due to computation precedence constrains, conflicts in memory access, conflicts in communication paths, or many other reasons.

Parallel computing systems are characterized by three main factors that highly affect their performance. These are processor architectures, topology (interconnection network), and the operating system.

Dr. John Worlton of the United States Los Alamos Scientific Laboratory said once, "The designers of supercomputers will do better at exploiting concurrency in the computing problems *if they use a small number of fast processors instead of a large number of slower processors*" [19]. This comment, indeed, agrees with reported studies. In addition it is much easier to operate and control homogeneous systems, because obviously this requires simpler scheduling strategies which are easier to design and code. Processors topology has a major impact on the routability issues of parallel systems and thus on the delivery span times of processor interchanged messages. Ring, hypercube, and fully connected are some examples of processor interconnection networks.

Operating systems play the role of assigning the given computation tasks to the set of available processors; this is what is called scheduling. It is crucial for the parallel system to have an efficient scheduler, since the scheduler largely determines the efficiency of the whole system.

Chapter 2

Problem Definition and Presentation

Earlier research work to solve the scheduling problem is of little practicality for current available technologies and systems. However, it has composed a robust platform of basic theories and strategies for research work that follows.

This chapter starts by defining the models used in the literature to represent parallel computations and multiprocessor systems. Then the nature of the scheduling problem is discussed. Finally, we will state our objective and the related assumptions to solve the scheduling problem.

2.1 The Scheduling Problem

Generally, a parallel computation can be viewed as a finite set of partially ordered computational tasks that may exchange messages among each others, i.e. as a set $\Gamma = (T_1, T_2, ..., T_m)$ of m tasks along with their computation times, precedence constrains, and inter-task communication requirements. Given a parallel computation of m tasks and a multiprocessor system of n identical processors, the scheduling problem can be defined as mapping the set of computational tasks to the available processors so that the finishing time of the resulting schedule is minimum while the precedence constrains are preserved.

2.2 Model Definitions

There are two models that are used to represent parallel computations. The first model, which is assumed in this thesis, considers the inter-task communication costs, and thus the topology of the multiprocessor system. This model was first proposed by Hwang [18]. The second model, used by Graham [15], assumes no communication overhead among tasks, and thus no message passing between processors. This model is presented here just to clarify any reference made to it through the literature review chapter.

A parallel computation with communication can be modeled as a directed acyclic graph (DAG), such that each node in the graph stands for some task, $T \in \Gamma$, in the

parallel computation, and each edge in the graph represents a precedence relation between a task and its immediate successor. Edges are labeled with non-negative integers $c(T, T_i)$ which express the number of message units to be sent from some task T to its immediate successor T_i, upon the completion of the task T, and the set of inter-task communication costs is denoted by C. Edges are assumed always to be directed towards the bottom of the computation graph, and numbers inside nodes represent task computation times ($\mu(T)$). Without loss of generality, we can assume that our computation graph has one entry node, i.e. a node with no (zero) predecessors, and only one exit node , i.e. a node with no (zero) successors. This is always possible, since we can add dummy nodes and edges to convert any graph to the required form. Also, we can assume that each node is reachable from the entry node, and can reach the exit node. So, a general precedence constrained computation with communication costs can be denoted by the quadruple $G(\Gamma, \rightarrow, \mu, C)$, and we call it the task model. [18] [3] [6] [4]

The delivery time of the $c(T, T_i)$ messages is dependent on the communication media and processor topology. The multiprocessor system is denoted by the pair $S(P, R)$, and we call it the system model. P represents the set of n identical processors and $r(p, p')$ is the cost in time to send a message unit between the two processors p and p'. The set of inter-processor routing costs is denoted by R. By varying the values of μ and c one can use $G(\Gamma, \rightarrow, \mu, C)$ to express graph problems with arbitrary task computation times and inter-task communication costs. In addition, by

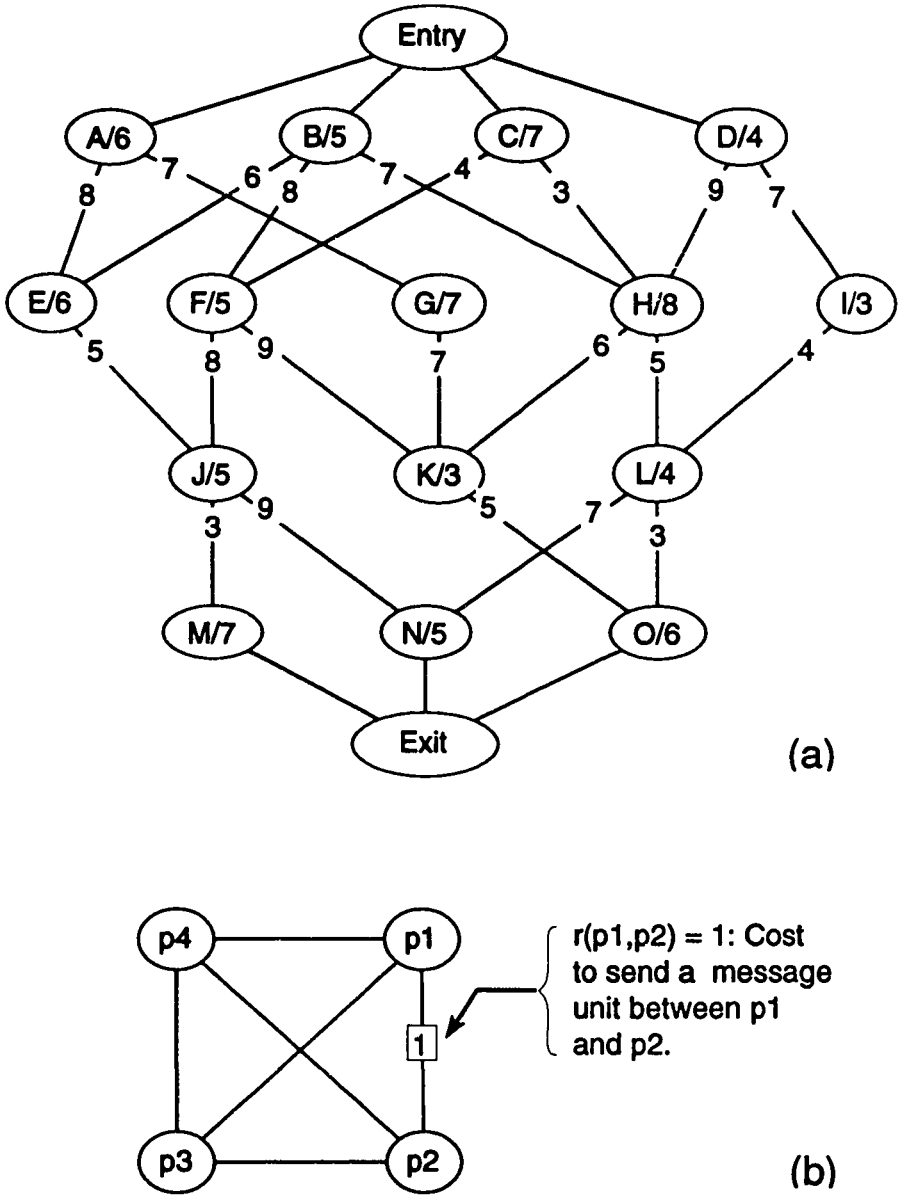

Figure 2.1: Example of (a) A Precedence Graph with Communication Costs (b) A Multiprocessor System

adapting the values of the function r, various multiprocessor systems, with various degree of connectivities, that span from tightly coupled systems to loosely coupled systems, can be modeled. Figure (2.1) shows an example of a parallel computation with communication costs $G(\Gamma, \rightarrow, \mu, C)$ modeled as a directed acyclic graph and a multiprocessor system $S(P, R)$.

There are four functions that are of main interest here. The first one, denoted by $\mu(T)$, takes one input parameter T of type task, and returns the computation (execution) time of T. The second function, denoted by $p(T)$, again takes one input parameter T of type task, and returns the processor to which task T has been assigned. $c(T, T')$ is the third function. It takes two input parameters, T and T', of type task, and returns the number of message units to be transmitted form task T to task T' upon the completion of T. In fact, $c(T, T')$ is meaningful for immediate tasks only [1], so if T and T' are not immediate tasks, then $c(T, T')$ will return 0 always. The forth function, denoted by $r(p_1, p_2)$, again takes two input parameters, p and p', of type processor, and returns the cost to send a message unit between the two processors p and p'.

The second computation model presented here is actually a special case of the first model and is denoted by $G(\Gamma, \rightarrow, \mu)$. Inter-task communication costs are not considered in this model, and thus the processor topology and routing times are not of interest in this task model. The resources here are m tasks, with their computation

[1] Two tasks T and T' are said to be immediate task if T' is a direct successor of T.

times, and precedence constraints which are to be scheduled on n identical processors in order to minimize the resulting schedule overall finishing time. [15] [12]

2.3 Common Terms

In the following we will define some terms that are frequently used in the scheduling context. In fact, the definitions of some of these terms depend on the definition of the path length between any given two tasks, which in turns depends on the computation model assumed.

- **Path Length**: For the model $G(\Gamma, \rightarrow, \mu)$ that assumes no communication between tasks, the path length is a static quantity as it is independent of the task-processor assignments and can be evaluated from the computation graph only. In this case, the length of a path between two tasks T and T' is defined as the sum of task computations along that path.

  However, for the model $G(\Gamma, \rightarrow, \mu, C)$ where inter-task communication costs are considered, the evaluation of path length should incorporate the communication delay among tasks. This is, therefore, a dynamic quantity as it is dependent on the task-processor assignments. Consider the two immediate tasks T and T'. The communication delay from T to T' is a function of the routing cost $r(p(T), p(T'))$ from $p(T)$ to $p(T')$. We can distinguish here between three situations. The first one is when T' is assigned to run on the

same processor of task T, that is $p(T)$, then no communication delay is required since the $c(T,T')$ message units are available on the private resources of processor $p(T)$. The second situation is when T' is assigned to run on another processor, $p(T) \neq p(T')$, then $p(T)$ has to send $c(T,T')$ message units to processor $p(T')$, and the communication delay is equal to the number of transmitted message units multiplied by the cost to route a message unit between the two processors , that is $c(T,T') \times r(p(T),p(T'))$. The third situation occurs when T' is not yet scheduled, i.e. assigned to some processor, or both T and T' are not yet scheduled, then the communication delay from T to T' is considered to be equal to the number of message units $c(T,T')$ to be sent from task T to task T'. For example, consider the computation graph and the multiprocessor system of figure (2.1). Assumes that tasks B and F have been assigned to the same processor, i.e. $p(B) = p(F)$, and J has not scheduled yet, then the path $B \rightarrow F \rightarrow J$ has a path length of $\mu(B)+c(B,F)\times r(p(B),p(F))+\mu(F)+c(F,J)+\mu(J) = 5+0+5+8+5 = 23$ time units.

Another approach to evaluate the path length for this model, is to add the values of the communication edges to task computations along that given path [11]. This approach is a static one as the evaluation of path length is independent of task-processor assignment and can be computed from the computation graph directly . Note that there may be more than one path

between T_i and T_j, each having its own path length.

- **Critical Path**: The critical path of a given computation graph G is denoted by CP and is defined as the longest path from the entry node to the exit node. For the model $G(\Gamma, \rightarrow, \mu)$, the critical path is again a static quantity that can be found from the graph only. While for the model $G(\Gamma, \rightarrow, \mu, C)$, the critical path is a dynamic quantity that can change during the scheduling process. This is because of the communication delay among tasks as explained above. For example, the critical path of the computation graph in Figure (2.1) before starting the scheduling process is the path along tasks $\{B, F, , N\}$ and has a path length of 45 time units. Later during the scheduling process assume that the entry tasks $\{A, B, C, D\}$ have been assigned to the available four processors in the multiprocessor system of figure (2.1), and that task F has been assigned to the same processor where task B was assigned, that is to $p(B)$. In this case, the communication delay between tasks B and F is zero and the path $B \rightarrow F \rightarrow J \rightarrow N$ will have a path length of 37 units. However, the path $A \rightarrow E \rightarrow J \rightarrow N$ is the critical path of this partial schedule and has a path length of 44 time units.

- **Task Level**: The level of a task T with respect to a given computation graph is denoted by $l(T)$ and is defined as the longest path from the task to the exit node. For the model $G(\Gamma, \rightarrow, \mu)$, task level is a static quantity that can be eval-

uated for each task from the graph only. While for the model $G(\Gamma, \rightarrow, \mu, C)$, the task level is a dynamic quantity that can change during the scheduling process. This is due to the communication delay among tasks. Again consider the computation graph and the multiprocessor system of figure (2.1). Assumes that neither task E nor any of its descendants J, M, N have yet been scheduled. So the level of E is 30 units, that is the length of the longest path from E to the exit node ($E \rightarrow J \rightarrow N$). During the scheduling process, later, assumes that $p(E) = p_1$, $p(J) = p_2$, $p(N) = p(J)$, and $p(M) = p_1$, then the level of E becomes $l(B) = \mu(E) + c(E, J) \times r(p(E), p(J)) + \mu(J) + c(J, M) \times r(p(J), p(M)) + \mu(M) = 6 + 5 \times 1 + 5 + 3 \times 1 + 7 = 26$.

- **Ready To Run Tasks**: A schedulable task or *ready to run task* is a task that has all its predecessors already assigned to some processors. At any time during the scheduling process, ready tasks are grouped in a set called the set of ready-to-run-tasks, and is denoted by R.

- **Task Earliest Starting Time** : A task T can not start its execution on an available processor p until all its predecessor tasks, $T_i \in pred(T)$ have completed their executions, and processor p has received the last message to T form its predecessors. The earliest possible time a task T can start its execution, is called the *earliest starting time* of T and denoted by $est(T)$ [2].

[2]More details can be found in chapter 3.

- **Completion Time:** Given a task T, the sum of T's earliest starting time ($est(T)$) and its execution time ($\mu(T)$) defines what is called *earliest completion time (ct(T))* of T. So $ct(T) = est(T) + \mu(T)$.

2.3.1 The nature of the scheduling problem

The scheduling problem in its most general form was proved to be NP-complete [9] [32] [14]. In fact, the problem is NP-complete even in the simple cases of scheduling a set of one-unit or two-unit tasks on a system of two processor [9], and of scheduling a unit-time tasks to system of arbitrary number of tasks [16]. In the literature, various relaxing assumptions regarding the computation graph structure and the multiprocessor system, have been made in order to simplify the problem. For example, when inter-task communication is not considered, the following are two exceptional cases where optimal schedules could be found in polynomial time:

1. scheduling computation graphs of arbitrary precedence structures with equal task execution times on a system of two processors [8] [28],

2. scheduling tree-structured computation graphs with equal task execution times on any number of processors [17].

Both of the optimal algorithms, start by evaluating a task priority measure for all tasks using the task level concept which proved the effectiveness of using global information. If inter-task communication is considered, then the evaluation of task

priorities should incorporate the communication delay among tasks which is a dynamic factor that can be only estimated during the scheduling process after assigning tasks to processors. This is because the communication cost between two tasks is considered zero if they are assigned to the same processor, or it is equal to the number of transmitted messages multiplied by the cost to send one message unit between the processors if the communicating tasks has been assigned assigned to two different processors. It was proved that introducing the communication of any of these two cases makes the problem an NP-complete one [11], Further, Prastein [26] proved later that scheduling computation graphs of arbitrary precedence structures with equal task execution times and inter-task communication, on a system of two processors is an NP-hard problem.

For the purpose of the dataflow architectures, Lee and Bier [23] introduced a scheduling taxonomy that is worth to mention here. They distinguished between four types of scheduling. The first type is the Fully Dynamic Scheduling where a task that has all its messages available is assigned to an idle processor at run time only. The second type is the static assignment scheduling where tasks are assigned to processors at compile time, and a run-time local scheduler should decide later their relative fire times. The third type is called self-time scheduling. A linear order among tasks, assigned to the same processor, is determined at compile time. At the execution time, each processor waits for the availability of the input messages for the next task in its ordered list. The last scheduling type is referred to as fully static,

where the compiler specifies task-processor mappings, the relative order among tasks on each processor, and task-fire times.

2.4 Static Scheduling

In this research we will deal only with static scheduling where task-computation times, precedence constraints, inter-task communication costs are known beforehand. This is in contrast to dynamic scheduling that lacks the global knowledge about tasks. Efficient static schedulers are important as compiler optimization techniques. The good thing about static schedulers is the ability to produce sub-optimal schedules, as they exploit the global properties of the given computation graph such as task level and critical path. However, static scheduling is unable to handle loops and branches [11] [10]. This is in contrast to dynamic scheduling as the exact computation time of a task that incorporates branches and loops can not be determined in advance before the execution phase of the given computation program [31].

Dynamic schedulers have to schedule tasks on the fly such as the scheduling decision is taken while the program is executing. The main deficiency of dynamic scheduling is the lack of global measure and the run-time processing overhead. The notion of load balancing is important in this context. This is due to the lack of a prior knowledge about the structure of the computation graph. So dynamic schedulers try to distribute the tasks fairly among the processors in a trial to maintain the

shortest schedule always.

2.5 Contention Free Systems

Contention occurs when two or more parallel tasks on the multiprocessor system want to send messages through common channels. Contention delays the arrivals of messages to their destinations. We are assuming that the commn channels of the multiprocessor system always have enough bandwidth to accommodate all the required communication with no delaying, and all processors have a support for temporal overlapped computation and communication activities. Such a system is called a contention free system. y, this implies that the time required to send n messages from a processor p to another processor p' is deterministic and always equals to $(n \times r(p,p'))$, where $r(p,p')$ is the cost o send one message unit from processor p to processor p'.

2.6 Non-Preemptive Scheduling

We are considering a non-preemptive scheduling. This means that once a task has started its execution on some processor, it can not be interrupted or stopped and must be allowed to run to its completion. In preemptive scheduling, however, the computation time of a task is sliced into a number of intervals and can be preempted and resumed several times on several processors. Further, each time the task resumes

on a different processor, its previous state from the last run should be migrated to the new processor. Generally speaking, this can increase the required communication in the system.

2.7 Research Objective

This research deals with deterministic scheduling of precedence-constrained computation with communication costs on distributed-memory multiprocessor. The computation is modeled by a directed acyclic graph in which a node represents the task computation time and a directed edge $T \rightarrow T'$ between tasks T and its successor T' is assigned a weight that represents the amount of data to be sent, upon completion, from T to T'. The objective function is to minimize the overall finish time of the computation graph over a given multiprocessor topology such as the fully-connected, hypercube, mesh, and so on. We will assume a latency-based model of the communications in the underlying multiprocessor. We also assume linear communication model which imply that communication conflict will not be considered in our work.

Our objective is to find an efficient scheduling strategy based on the concept of *Iterative Refinement Scheduling* (IRS) that has been previously developed [3] [5]. The basic idea of *IRS* is simple and involves alternatively scheduling the given computation graph in forward and backward passes, while passing the task completion time as the task level from one scheduling pass to another. Task level is a fundamen-

tal priority notion as it measures how much computation and communication are left from starting of the task to the end of the schedule (the exit node). Unfortunately finding the task level for precedence constrained computation with communication, as we will see in the chapter 3, is not an easy job. To overcome this problem, IRS approximates the task level by the task completion time resulted from the previous scheduling pass. This is because the task completion time resulted from a backward scheduling approximates the task level for the forward scheduling, and vice versa.

On the other hand, IRS uses priority-based criterion that is combined with management of the processor idle time. We believe that the IRS concept can produce better results if the task level was defined in a more refined manner. Our objective can be summarized as :

1. Find a more refined strategy that can be incorporated with in the IRS concept, to approximate the task level.

2. Investigate the efficiency of combining the refined approximation of task level with management of processor idle time, in order to find more efficient combination of these two important quantities.

Chapter 3

Literature Review

This chapter presents a review of some of the newly reported scheduling heuristics and strategies, and bounds on the scheduling problem. It also summarizes heuristic general behaviors. We shall distinguish between two scheduling control approaches that are going to be introduced in the following section. Then we will start our review of scheduling heuristics and strategies with priority-based scheduling. In Sub-Sections 3.2.1, 3.2.2, and 3.2.3 we present List Scheduling, Generalized list scheduling, and the Iterative Refinement Scheduling, respectively. Then, in the subsequent sections, we continue the review with other relevant scheduling heuristics. We present our review of bounds on scheduling in the last section .

3.1 Scheduling Control Approaches

Graham introduced one approach to control the scheduling process that will be referred to as processor driven [15]. Another approach for controlling the scheduling process is called computation driven (CD) [6] [3].

3.1.1 Processor driven

In processor driven scheduling control approach (PD), the set of ready-to-run tasks is modified when any processor has finished execution of some task T and becomes idle. The set of immediate successor tasks of T denoted by $succ(T)$ is checked to see if any of them is ready, to be add to the set of ready-to-run tasks [6] [15]. When PD control approach is used in the scheduling process, expansion through the computation graph is more breadth oriented than depth oriented, without emphasizing the notion of path criticality by giving comparable opportunities to all paths. This control approach results in a uniform scheduling of processors since the starting time of successively scheduled tasks form a non-decreasing sequence in time. The processor driven scheduling control approach, in fact, is derived by processor completion times.

3.1.2 Computation driven

The basic difference between computation driven CD control approach and PD, is that when a task T is assigned to a processor p, the set of ready-to-run tasks is modified immediately to incorporate any immediate successor of task T whose other predecessors have all been assigned to some processors but not necessarily have completed their executions. This control approach results in a more vertical expansion of the computation graph G. Further, CD approach emphasizes the notion of critical path, and allows the scheduling process to investigate the computation graph in depth, while PD does not. This gives computation driven based scheduling heuristics, in general, the chance to select a task for assignment among a larger set of ready-to-run tasks, with much more aberration of task level values. So it seems very logical for computation driven based scheduling heuristics to incorporate this new factor, aberration of tasks level values, in their scheduling decision functions. One way to do this is to use some non-increasing function of task level values to discriminate critical tasks from less critical ones in the set of ready-to-run tasks.

One can look to processor driven or computation driven as just scheduling control approaches. Their effects, in the scheduling process, are in determining when candidate tasks are to be considered ready and included in the set of read-to-run tasks. For PD control approach, by modifying the set of ready to run tasks at the completion of a task execution, it is imposing a breadth more than a depth

traversing of the computation graph. So it is not a pure breadth wise and some conflict situations can arise [18]. However, CD allows the modification of the set of ready-to-run tasks to be just after the assignment of tasks to processors. This gives the scheduling process the opportunity to go in a deeper manner more than PD does.

3.2 Scheduling Techniques and Strategies

Most of the research done today to solve the scheduling problem is revolving around enhancing the classical heuristic methods, applying optimization techniques on previously reported heuristics, or apply techniques mainly used in other areas like artificial intelligence. For example, branch and bound techniques have been used to reduce the search space of feasible schedules [20]. Several brach selection strategies have been investigated [24]. In fact, this approach is suitable for small graph problems where the number of branches generated for each branch node is relatively small.

The main concern here is construction-based scheduling algorithms. We will introduce and discuss various recently reported scheduling algorithm. This is in a trial to explain their main operational trends along with their deficiencies and limitations.

3.2.1 Priority based scheduling and Graham's list scheduling

Priority based scheduling utilizes a selection strategy to pick a task among a set of ready-to-run tasks and assign it to an idle processor in an attempt to minimize the overall finishing time of a given set of computations [30]. We can distinguish between two classes of selection strategies. These are local selection strategies like Earliest Task First ETF heuristic [1], and global scheduling strategies like Highest Level First HLF [2]. Local selection strategies try to minimize processor idle times as a mean to minimize the overall schedule finishing time. Also, local strategies lack the utilization of global information. This is in contrast to global strategies where the notion of task level is fundamental.

List scheduling

Graham [15] introduced one of the fundamental results related to the scheduling problem, that is list scheduling LS and its performance guarantee bound. The objective is to minimize the overall finishing time of a partially ordered set of computation tasks with no communication on a multiprocessor system. List scheduling stores the tasks in a non-decreasing order of their task level values in a priority list L. At each scheduling step, the list L is scanned from the beginning and the first

[1] At each scheduling step, ETF schedules the ready task that has the earliest starting time.

[2] At each scheduling step, HLF schedules the ready task that has the largest level value, to the available processor that reduces its starting time.

ready task T is fetched and assigned to an idle processor to run without preemption. The set of ready-to-run tasks is updated at the completion of T, to include any of its immediate successor tasks that became ready. This process continue until all the tasks have been scheduled.

The evaluation of task level values for the computation model with no communication $G(\Gamma, \rightarrow, \mu)$ can be done quickly by starting with the exit node and tracking backwards up to the entry node [17] [15]. In fact, the definition of task level for this model is independent of task-processor assignment and can be evaluated from the computation graph only. Let $succ(T)$ be the set of immediate successors of a task T, the task level $l(T)$ can be computed as:

$$l(T) = \mu(T) + \begin{cases} 0 & \text{if } succ(T) = \phi \\ Max\{l(T_i) : T_i \in succ(T)\} & \text{otherwise.} \end{cases} \tag{3.1}$$

There are two important aspects of list scheduling. The first one is that list scheduling uses a decision function that implicitly enforces the starting times of successively scheduled task to form a non-decreasing sequence in time. In fact the worst case bound of list scheduling [15] was derived from such enforcement rule, and regardless of the method used in task level evaluation. The second important aspect of list scheduling is the use of task level as a measure of task criticality. So that at any time a processor is free, it immediately begins to execute the ready task which currently heads the longest chain of unexecuted tasks. List scheduling can generate near optimum solutions. Empirical testing [1] proved in a statistical manner that

LS generates solutions that are just 5% away from the optimum solution in 90% of studied cases.

In memory shared systems, it is fair to refer to scheduling precedence computation graphs with negligible inter-task communications, since the computation time not the communication cost is the dominating factor in the resulting schedule finishing time. However, inter-task communications plays an important role for scheduling distributed memory systems (message based systems) and cannot be neglected.

Although the task level notions allows the discrimination of critical tasks, the evaluation of task level for precedence computation graphs with communication costs is very expensive. This is because the evaluation of task levels in this case is dependent on the task-processor assignments which are not known beforehand, and requires the knowledge of the optimum finish time. The fact that the evaluation of a task level depends on the task processor assignments, is because the communication delay between two immediate tasks T and T' is considered zero if both have been assigned to the same processor. While in the case the two tasks have been assigned to different processors, $p(T) \neq p(T')$, the communication delay is equal to the number of transmitted messages $c(T, T')$ multiplied by the by the cost to route a message unit from processor $p(T)$ to processor $p(T')$. Further, the problem of optimally scheduling graph problems with communication, as mentioned above , was proved to be NP-complete.

As it is, list scheduling can not be applied directly for scheduling precedence

computation graphs with communication costs, because its way to evaluate task level values does not consider the communication delay among tasks. So heuristic strategies should be explored for evaluating the task level, in order to enable the design of efficient global scheduling algorithm aimed for the model $G(\Gamma, \rightarrow, \mu, C)$.

By extending the definition of task level above in equation 3.1 to account for the communication edges along a path from T to the exit node, we will get:

$$l(T,p) = \mu(T) + \begin{cases} 0 & \text{if } succ(T) = \phi \\ Max\{l(T_i) + c(T,T_i) : T_i \in succ(T)\} & \text{otherwise.} \end{cases} \quad (3.2)$$

However, accounting for communication edges in such a static manner with out considering the topology latency of the multiprocessor system, dose not yield a realistic approximation of the task level. This is because the routing delays among processors are not identical. Imagine the inaccuracy introduced for the approximated task level values when some high communication edges are zeroed during the scheduling process.

Both, discarding completely the effects of the communication delay among tasks [21] [25] [29], or pessimistically accounting for all the communication requirements [11] [33], in the evaluation of the task level can result in inaccurate estimations of the criticality of tasks with respect to the whole computation graph. In fact, a realistic estimate of the task level for the model $G(\Gamma, \rightarrow, \mu, C)$, should consider task computation times, communication edges, and the routing latency of the multiprocessor system and thus records for the so far done task-processor assignments.

Let T' be a predecessor task of a given task T, i.e. $T' \in pred(T)$ where $pred(T)$ denotes the set of all immediate predecessor tasks of T. The earliest starting time $est(T,p)$ of T on some processor p depends on:

1. the completion time $ct(T')$ of T' on $p(T')$,
2. the number of message units $c(T',T)$ to be sent from T' to T, and
3. the cost to send one message unit form $p(T')$ to $p(T)$, that is $r(p(T'),p(T))$.

So, we can write the earliest starting time of a given task T on some processor p as :

$$est(T,p) = ct(T') + c(T',T) \times r(p(T'),p).$$

Now, by considering all the predecessor tasks, $est(T,p)$ is the earliest starting time of task T on processor p provided that all the precedence constraints have been satisfied and by which all the required communication has reached processor p.

$$est(T,p) = Max_{T' \in pred(T)}\{ct(T') + c(T',T) \times r(p(T'),p)\}.$$

If $pred(T) = \phi$, then $est(T,p) = 0$. This definition may not be exactly achievable as p might be busy executing some other task. Let $eft(p)$ denotes the earliest time p becomes free, the effective earliest starting time of T on p should be rewritten as:

$$est(T,p) = Max[Max_{T' \in pred(T)}\{ct(T') + c(T',T) \times r(p(T',p)\}, eft(p)]$$

Note that the definition of $est(T,p)$ depends on the routing cost $r(p(T'),p)$. As a result, there must exist a processor p^*, among all the available processors in the multiprocessor system, that reduces the starting time of T to its earliest time:

$$est(T) = est(T,p^*) = Min_{p' \in P}\{est(T,p')\}.$$

Further, the completion time $ct(T)$ of task T is defined as $est(T) + \mu(T)$. In fact, the $ct(T)$ of task T approximates the shortest possible distance from the entry node to the completion of T provided that all the precedence constraints and the communication requirements are satisfied.

Generalized List scheduling

A heuristic estimate of task level values that counts for computation, communication, network latency, and records for so far done task-processor assignments was proposed by Al-Mouhamed and Al-Massarani [3] [6]. Let G_r denotes the reverse graph of $G = G(\Gamma, \rightarrow, \mu, C)$ obtained by reversing the directions of all edges in G. An estimate of task level values can be obtained by scheduling G_r using the best local scheduling heuristic ETF on the multiprocessor system $S(P,R)$. This is called a backward scheduling pass as the scheduling of G_r begins with the exit node of G and continues up to the entry node. The task completion time $ct(T)$ obtained from the scheduling of G_r, approximates, for the original graph G, the remaining computations and communication times along a chain from the starting of task T to

the exit node of G. So the completion time $ct(T)$ of a task T resulting from a backward scheduling pass, approximates the task level of T for scheduling the original computation graph G on the multiprocessor system S. The next step is to use the approximation evaluation of tasks level values, i.e. $l(T) = ct_{back}(T)$, to schedule the computation graph G according to some global strategies in an attempt to minimize the over all finishing time .

Generalized list scheduling (GLS) is a class of scheduling heuristics that uses global information in order to schedule precedence graphs with communication costs on multiprocessor systems [6] [3]. This is a generalization of Graham's list schedulling where inter-task communication costs and processor topology are considered. Scheduling heuristics that belong to this class consists of the following steps:

1. *Initialization*: evaluate $l(T)$ for all tasks in the given computation model $G(\Gamma, \rightarrow, \mu, C)$ on the system $S(P, R)$.

2. *Scheduling* : until there is no unscheduled task, select the ready to run task with the highest global decision function to start on some idle processor.

A Computation-Driven/Highest-Level-Earliest-Task-First ($CD/HLETF^*$) [6] [3] is one representative of this approach. This heuristic takes the task completion time resulting from a backward scheduling pass as task level for scheduling the given computation graph. More accurately, its decision function is $l(T) - est(T)$, where $l(T)$ is the completion time that resulted from the backward scheduling pass and

$est(T)$ is the earliest starting time in the forward pass. The combination of $l(T)$ and $est(T)$ was considered as an attempt to improve the management of idle time within the framework of GLS.

Iterative refinement scheduling

It is obvious that the approach of approximating the task level by the task completion time is based on one step of backward scheduling pass to schedule the G_r, followed by one step of forward scheduling pass to schedule G. Iterative Refinement Scheduling IRS is a scheduling approach that iterates this process several times, in order to enhance the approximation of the task level. So the IRS scheduling consists of alternatively scheduling the forward and backward task graphs by using a global scheduling heuristic and passing the task completion time $l(T)$ that results from one pass to the next pass [3] [5]. The heuristic $CD/HLETF^*$ was further improved by using it within the framework of such iterative process.

One important property of the iterative process is that of having subsequent high-low schedule finishing time values. In fact, the general trend of IRS is to find better schedules in the first few iterations, however the improvement process will get slower as finding better schedules become harder.

This approach proved to be useful in refining the initial solution. Further, the behavior of IRS can be one of the followings:

- The first situation is where IRS converges to its best solution, i.e. the resulting

schedule finishing time is the minimum value that IRS has come across.

- The second case is where IRS converges to a value deviating a little from its best solution.
- The third case is where IRS process oscillates, and does not converge to a specific value.
- The last observation is when IRS process does not converge to a specific value within the predetermined number of iterations.

Moreover, empirical testing [3] [5] showed that the improvement of IRS is not due to exploring possible schedule permutations and finding arbitrary better ones, rather it is a deterministic process. Actually, IRS was applied using a random heuristic that selects randomly a task from the set of ready-to-run tasks and assigns to the idle processor that can reduce its starting time. One can visualize this as a random walk through the possible scheduling search space. Though IRS was able to enhance the single random iteration scheduling and produce schedules with lower overall finishing time, these results were far away from being acceptable. Further, when IRS was applied using $CD/HLETF^*$ with random task-level values at the first iteration, it was able to recover quickly and find more accurate task-level values and converged to the same finishing time that is generated using deterministic start, but required more iterations.

3.2.2 Dynamic level scheduling

Dynamic Level Scheduling [29] utilizes a dynamic decision function an assumes a bounded number of processors. The dynamic level value of a given task is evaluated using two quantities, the first one is the static level of the task (SL) which is the largest sum of computations from the task to an exit node. The second quantity is a dynamic one, that is computed at each scheduling step for all unscheduled tasks. This is the start time $ST(x,p)$ of task x on a processor p. At each scheduling step, the priority for every ready task, denoted by dynamic level, is computed as:

$$DL(x) = SL(x) - ST(x),$$

and the task-processor pair that contribute the largest DL value is selected so that the task is assigned to the processor. DLS has a high time complexity of order $O(v^3pf(p))$, where $pf(p)$ is the complexity of calculating the ST of a node at each step. The scheduling decision function of DLS has the drawback of penalizing the more accurate task start time, by the less accurate static level value which measure the task criticality.

3.2.3 Task clustering

The clustering strategy of Sarker [27] suggested a two step scheduling approach. These are:

1. schedule the given computation graph on unlimited number of fully connected processors of comparable abilities. This will result in clusters of tasks, such that each cluster has been scheduled to the same processor,

2. if the clusters counts more than the number of physical processors, then these clusters have to be merged to the number of physical processors in a manor that accounts for the topology of multiprocessor system.

Sarkar's algorithm for the first step is called **Edge Zeroing (EZ)**. This heuristic sorts the edges of the given computation graph in a non-increasing order of their communication costs. At each scheduling step, EZ selects the two tasks with the heaviest communication cost and assign them to the same processor if the resultant partial schedule is the shortest, otherwise the two tasks are scheduled to two different processors so that the resultant partial schedule length is the shortest possible. For this heuristic to preserve the precedence constraints among tasks, it assures that tasks assigned to the same processor are maintained in a non-increasing order of their task level values [3]. EZ has a time complexity of $O(e(e+v))$, where e is the number of edges and v is the number of tasks in the given graph. In fact, the general trend of EZ is to minimize the communication instead of exploiting the inherited parallelism of the given computation graph.

[3]The task level in this context does not consider communication edges, and is defined as the largest sum of computations along a path from the task to an exit node.

Yang and Gerasoulis [34] proposed another algorithm for the first step also, and is called **Dominant Sequence Clustering (DSC)**. DSC starts by computing the priorities for all tasks in the given computation graph. To do this DSC defines the top level and bottom level for each task. The top level of a task T, denoted by *tlevel(T)*, is the largest sum of computation and communication from an entry node to T excluding $\mu(T)$. Further, the bottom level of a task T, denoted by $blevel(T)$, is defined as the largest sum of computation and communication from the starting of T to an exit node. The dominant sequence DS for a partially scheduled graph is the path with the largest sum of computation and communication in the graph. A Task priority is defined as

$$PRIO(T) = tlevel(T) + blevel(T). \tag{3.3}$$

In fact, DS-tasks have the highest priority values. Tasks that have their parents already scheduled are referred as free task. This is in contrast to a partially free task that does not has all its parents already scheduled.

At each scheduling step, DSC picks the free task with the highest priority, let x denotes such a task, and assign to the processor that most minimize its *tlevel* value. This is unless this processor has been assigned some task that is the parent of one partially free DS-task y. In this case no task is allowed to be scheduled to this processor before y is free, and x is scheduled to another processor that can produce the least $tlevel(T)$. This is what is called Dominant Sequence Reduction Warrantee

(*DSRW*). *DSRW* is to guarantee effective reduction of *tlevel* values of partially free DS-tasks at the future scheduling steps.

Though *DSC* is able to identify the most critical task at each scheduling step, it does not schedule it until it is free which may end up in delaying some *CP* tasks. Another disadvantage with *DSC* is that it schedule a task to to new processor if it is the case that its *tlevel* value can not be reduced by scheduling it to any of the so far used processors. This can result in *DSC* utilizing more processor than necessary. *DSC* has a time complexity of order $O((e + v)logv)$, where e is the number of communication edges and v is the number of tasks.

3.2.4 Dynamic critical path

Dynamic Critical Path [22], denoted by *DCP*, has a dynamic scheduling decision function, and assumes unbounded number of processors. It is known that during the scheduling process the *CP* of a partial schedule of a given computation graph is not constant, so it is going to be referred as a dynamic critical path. For this scheduling heuristic to minimize the resulting schedule length, it selects at each scheduling step the DCP-task that has no unscheduled parent that is a DCP-task. For *DCP* to identify *DCP*-tasks , it evaluates two attributes for each unscheduled task. The first attribute is called absolute earliest starting time (*AEST*) and can be evaluated by traversing the given computation graph in a breadth-first manor starting from the entry task. In this sense, the length of the current (dynamic)

critical path length, denoted by $DCPL$, is defined as

$$MAX_i\{AEST(x_i, p(x_i)) + \mu(x_i)\}$$

$DCPL$ is used as a reference to compute the second attribute, that is absolute latest start time ($ALST$). The $ALST$ for each unscheduled task can be computed by traversing the graph in a breadth-first manor but in the reverse direction starting from the exit node. DCP-tasks are those tasks that have equal $AEST$ and $ALST$ values. DCP heuristic schedules tasks to processors without assigning them fixed start times until all tasks are scheduled. In fact, DCP just clusters tasks over processors in a linear order. By not fixing the start times, more critical task considered in later scheduling steps can be inserted into earlier time slots by just modifying the $AEST$ and $ALST$ values of the already scheduled tasks on a processor. This processor selection approach, in fact, is a requirement for DCP to guarantee the precedence constraints. This is because the decision function of DCP is designed in a way that does not pay attention if the selected task is ready or not. DCP heuristics uses also a look ahead strategy as part of its processor selection approach. The selected task x, at each scheduling step, is assigned to the processor p that lowers the value of $AEST(x,p) + AEST(x_c,p)$, where x_c is the most critical child of x and $AEST(x,p)$ is computed after tentatively assigning x to p. DCP has a relatively high time complexity of the order $O(v^3)$, where v is again the number of tasks in the given computation graph.

3.2.5 Task duplication

The communication overhead between tasks assigned to different processors can generally be reduced by executing redundantly some tasks that some other critical tasks depend on. Scheduling heuristics that incorporate this idea of redundant execution are referred as duplication-based scheduling heuristics. The first algorithm to consider has been proposed by Kruatrachue and Lewis [21],and is called Duplication Scheduling Heuristic (DSH). *DSH Algorithm:*

11) Repeat

2) Constructs a list of processors (PL) that includes all the processors to which the predecessor tasks of the most critical task, referred as the candidate task (CT), are scheduled, along with the processor that has the earliest free time.

3) Repeat for each processor p in PL:

3.1) Evaluates *idle time slot*, of processor p, as the difference between the finish time of the last task that has been already scheduled to p, and the earliest start time of the candidate task on p.

3.2) Evaluate the earliest start time of the dominant predecessor [4] of the candidate task (D_{CT}) on p. If $est(D_{CT}, p)$ is with in the *idle time slot* of p, duplicate it in this slot. If the *idle time slot* can accommodate this

[4]The predecessor task which sends the data that arrived last.

predecessor task, apply steps **3.1)** and **3.2)** to this dominant predecessor. However, If the dominant sequence overflow the *idle time slot* or the earliest start time of CT is not lowered, the duplication process for this p is terminated, and the original situation is restored.

3.3) Repeat step **3.2)** for the next dominant predecessor task of the candidate task.

4) Assign the candidate task to the processor that generate the least start time. Until all tasks are scheduled.

Bottom-up-top-down Duplication Heuristic ($BDSH$) [7] is an extension to ($BDSH$) where the duplication process does not stop as far as the idle time slot has not yet overflown. This is regardless of the possible increase in the start time of the candidate task. Further, $BDSH$ dose not give preference to any predecessor task to be considered in the duplication process. It is shown experimentally that $BDSH$ outperforms DSH for computation graphs with large ratio of communication to computation.

Booth DSH and $BDSH$ have a time complexity of $O(V^4)$, where v is the number of task in the computation graph. In fact, the application of task duplication for static scheduling, is not yet well explored. Ahmed and Kwok [2] proposed a more recent work on static scheduling applying this approach.

3.2.6 Modified critical path

Modified Critical Path [33], denoted by (MCP) assumes a bounded number of processors. This scheduling heuristic first computes the latest possible start time ($LPST$) for all tasks by delaying the start times for tasks as long as possible. This can be achieved by compressing the given computation graph as much as possible towards the exit node, so that this compression process is bounded by the length of the CP. Then MPC construct a list for each task that starts with $LPST$ value of the task it self followed by a non-increasing order of $LPST$ values of its children tasks. Further, MCP will order these lists in a lexicographical manor in another list called L. At each scheduling step, MPC selects the first task from the list L and assign it to the processor that can start it the earliest. The complexity of MCP is $O(v^2 log v)$, where v is the number of tasks in the given computation graph.

3.2.7 Mobility directed

Mobility Directed (MD) is another scheduling heuristic by Gajski [33]. It starts by computing the mobility (M) of each task as the difference between the task earliest possible starting time ($EPST(T)$), and the task latest possible starting time ($LPST(T)$). Then the relative mobility, denoted by M_r, is evaluated for a given task x as:

$$M_r(x) = M(x)/\mu(x)$$

This heuristic constructs a list (L') of tasks with minimum relative mobility. Then at each scheduling step, MD selects a task from L' that does not have any predecessor in L' and assigns it to the first processor that can accommodate this task. This can be done by pushing the already scheduled nodes, on this processor, downwards in order to create a large enough time slot. This is regardless of the possible degradation in the schedule length, In addition to the fact that choosing the first suitable processor does not mean that it is the best one. MD has a complexity of $O(v^3)$, where v is the number of tasks in the given computation graph.

3.3 Bounds on the Number of Processors and Schedule Finishing Times

Graham [15] [9] illustrated the scheduling anomalies that can arise in varying one of the parameters in $G(\Gamma, \rightarrow, \mu)$, including the priority list, that is a sequence of all the tasks, which is scanned repetitively by processors right away from the beginning, searching for a ready task.

It was found that changing the priority list to give higher priorities to tasks with high out-degree values, may increase the resulting schedule finishing time in some studied examples. Further, we would intuitively expect that relaxing the precedence constraints, decreasing task computation times, or increasing the number of processors, would cause the corresponding schedules finishing times to decrease if

compared to the schedule finishing times of the original computation graph before introducing the above changes. However, it was found that sometimes the above changes (introduced one at a time to the original computation system) cause the schedule finishing time to increase.

Graham also established one of the first upper bound on schedule finishing times. Factors that were considered in deriving the bound are the priority list, precedence constraints, task completion times, and the number of processors. Let ω denotes a schedule finishing time. The bound was constructed around the existence of a continuous chain of tasks that cover all the time intervals that belong to the schedule span time $[0, \omega)$, where at least one processor is idle. The general form of the bound is

$$\frac{\omega'}{\omega} \leq 1 + \frac{n-1}{n'},$$

where ω' is the resulting schedule finishing length after introducing the changes in all or some of the above parameters, and ω is the optimal schedule finishing time of the original computation system. It was proved that the bound is obtained in the sense that it cannot be replaced by any smaller function of the same parameters [15].

Hu [17] considered the two most important questions in the scheduling of multi processing systems:

1. How can one schedule a given computation graph G using a minimum number of processors and meet a predefined finishing time.

2. Given n processors and a computation graph G, find a schedule for G that finishes at the earliest possible time.

Hu first developed his general inequality that relates the number of processors to the resulting schedule finishing time:

$$p < \frac{1}{\gamma^* + c} \sum_{j=1}^{j=\gamma^*} \rho(\alpha + 1 - j),$$

where p is the number of available processors, $\gamma^* + c$ is a non-negative integer represents the time, α is the length of the critical path of G, and $\rho(\alpha = \alpha_i)$ is the number of tasks with $l(T) = \alpha_i$. Hu then solved the above two question for the case where the computation graph forms a tree structure, and the task computation times are equal.

A sharper bounds for the minimum number of processors and the minimum time was given by Fernandz [12]. He evaluated the completion interval for each task as the time interval between the task completion time and the task latest completion time. Then the activity of each task along time is determined accordingly. Fernandz then defined the load density function that indicates the activity of the whole computation graph for a given time interval ($[\theta_1, \theta_2]$). Let $R(\theta_1, \theta_2, t)$ denotes the load density function value after all tasks have been shifted within their completion intervals to reduce the overlap among them within this interval. A lower bound on the minimum number of processors required to execute a given computation graph that

has a critical path of length t_{cp} is:

$$P_L = \lceil \max_{[\theta_1,\theta_2]} [\frac{1}{\theta_2 - \theta_1} \int_{\theta_1}^{\theta_2} R(\theta_1, \theta_2, t) d_t] \rceil$$

where $\lceil \rceil$ is the least integer greater or equal to the value of this expression, and the minimum required number of processors is the maximum taken over all intervals. A lower bound on the minimum time required to execute a given computation graph on m processors have the general form of $t_L = t_{cp} + \lceil q \rceil$, where q is the increase in the overall schedule length over the critical path length (t_{cp}) because of not having enough processors to accommodate all the possible parallelism inherited in the given computation graph. q is evaluated by:

$$q = \max_{[\theta_1,\theta_2]} [-(\theta_2 - \theta_1) + \frac{1}{m} \int_{\theta_1}^{\theta_2} R(\theta_1, \theta_2, t) d_t]$$

where $m(\theta_2 - \theta_1)$ is the effective activity that can be performed with the m processors, and $\int_{\theta_1}^{\theta_2} R(\theta_1, \theta_2, t) d_t$ is the load so that the overall schedule length is t_{cp}.

List scheduling was later extended to incorporate inter-task communication costs. Hwang proved [18] that the schedule finishing time $\omega_{PD/ETF}$ satisfies the upper bound

$$\omega_{ETF} \leq (2 - \frac{1}{n}) \omega_{opt} + C,$$

where ω_{opt} is the finishing time of the optimal schedule, and C is the maximum communication costs along all chains in the given graph problem.

lower bounds, that evolved from Fernandz's load density function formulation, on the minimum number of processors and schedule finish times for scheduling prece-

dence graphs with communication costs were later proposed by Al-Mouhamed [4], where the interval $[\theta_1, \theta_2]$ was replaced by $[est_i, lst_i]$. He proposed an algorithm called Merge to compute the earliest starting time (est) for each task. Merge algorithm takes the task (T_j) to compute its est value along with the est's for its predecessors ($T_i : \forall i \in pred(T_j)$) . Merge tries to cluster as much predecessors as possible along T_j on the same processor so that $est(T_j)$ can not be reduced any more.

Chapter 4

The Proposed Scheduling Heuristic *CD/ERDETF*

In this chapter, we will review briefly our attempted approaches to achieve the defined objective. Then we will present our final heuristic approach to estimate the task level. Next, we will introduce our global scheduling heuristic *CD/ERDETF* along with its analysis and general behavior. Then we will explain how to incorporate the proposed scheduling heuristic in the iterative refinement process.

4.1 Attempted Approaches

Before we converged to our final strategy that will be explained in the next section, we attempted several approaches. Here, we are summarizing three main attempts.

Each scheduling iteration requires a backward scheduling pass followed by a forward scheduling pass. The main issue concerning the iterative approach, is the approximate evaluation of task level values. Inaccurate evaluation of $l(T)$ for some task T, can be grouped in one of the following two situations:

- when the task T is, in fact, a critical task but the evaluated $l(T)$ is relatively low, or
- when the task T is, in fact, a less critical task but the evaluated $l(T)$ is relatively high.

These situations will negatively affect the scheduling decisions in the next forward scheduling pass.

In our first attempt, we tried to use the distance from the entry node to the exit node from different iterations through a task T, as a criticality measure of T. At the end of each iteration, a new and a complete path along the computation graph from the entry node to the exit node, is approximated for each task. Assume that the iterative process is at its n^{th} scheduling pass, then $ct_{for}(T)$ and $est_{back}(T)$ approximate the path length $pl(T)$ for T in the current iteration. So, the path length $pl(T)$ of T can be defined as :

$$pl(T) = ct_{for}(T) + est_{back}(T).$$

Also, we have implemented another heuristic to estimate the path length. This is achieved by pulling the tasks towards the exit node in the resulted forward schedule,

and by pulling the tasks towards the entry node in the resulted backward schedule. In these situations, the path length $pl(T)$ of a given task T is defined as follows:

$$pl(T) = est(T) + (\omega - lst(T)),$$

where $lst(T)$ is the task latest starting time $lst(T)$ obtained by the pulling process, and ω is the schedule finish time.

We expected that the path length of some task T gives sharper criticality information than what its level value does. Empirical testing showed that the performance of this approach was by average 2% worse than that of $CD/HLETF^*$, for the studied cases. The lesson that we learned out of this approach, is that using the path length from the entry node to the exit node through a given task T as the task criticality measure is confusing the decision function, and what is important really is how much computations and communication costs is remaining from the T to the exit node.

In the second approach, we tried to alter the resulted task completion times before passing them to the next scheduling pass as task level values. In fact, the fluctuation behavior is very risky for the iterative scheduling process. The main problem of IRS can be summarized as:

- low $ct_{n-2}(T)$ [1] value for some critical task will result, most probably, in high $est_{n-1}(T)$ value (to reflect the actual length of its critical path) and this in

[1] The subscript denotes the scheduling pass number

turns will result again in low $ct_n(T)$, or

- high $ct_{n-2}(T)$ value for some less critical task will result, most probably, with low $est_{n-1}(T)$ value (to reflect the actual short path) and this in turns will result again in high $ct_n(T)$.

Even though one should expect that the approximate evaluation of $ct_n(T)$ value should be raised in the first situation and lowered in the second. A logical solution to resolve these situations is to modify the resulted task completion times obtained in the current scheduling pass, and forward the modified version to the next scheduling pass. So we may have something of the form

$$ct_n(T) = ct_n(T) \pm X. \tag{4.1}$$

Since high $est_{n-1}(T)$ value will result with small $ct_n(T)$ (problem 1) and the low $ct_{n-2}(T)$ value should be raised, then X is directly proportional to $est_{n-1}(T)$ and inversely proportional to $ct_n(T)$. So, we can rewrite the equation above as :

$$ct_n(T) = ct_n(T) \pm \frac{est_{n-1}(T)}{ct_n(T)}. \tag{4.2}$$

In fact the sign of X determines whether the resulted $ct(T)$ is to be raised or lowered. We used several strategies to determine the sign of X for each task. All of them considered the relation between the current schedule finish time ω_n, the current path length $pl_{curr}(T) = ct_n(T) + est_{n-1}(T)$, the schedule length of the previous scheduling pass ω_{n-1}, and the previous path length $pl_{prev}(T) = ct_{n-2}(T) + est_{n-1}(T)$.

Parall.	1	3
Better	62% by (3%)	18% by (1%)
Worse	32% by (2%)	82% by (2%)

Table 4.1: Performance of the Second Approach

Empirical testing showed that this approach is 3% by average better than that of $CD/HLETF^*$ in 62% of the studied cases, at low parallelism degree (see table 4.1). At relatively high parallelism degree, however, the performance of this approach is 2% by average worse than that of $CD/HLETF^*$ in 82% of the studied cases.

In the third approach, we keep for each task its so far discovered minimum completion time, and we continued to pass this value as the task level from one scheduling pass to another until we encounter a smaller completion time. In this case we update the level of this task to the new encountered smaller completion time and continue doing the same till the end of the iterative process.

This approach converged faster than $CD/HLETF^*$. Empirical testing showed that this approach required less than ten iterations to converge to a solution that is 3% by average worse than $CD/HLETF^*$.

4.2 The Limited Resources and the Serialization Effects

Graham used a method to evaluate the task level that starts by computing the task level of the exit node and propagates the accretion of computations up to the entry

node [15, 17]. This method can evaluate task level values $l(T)$s that do not increase the optimum schedule of a given computation graph with no communication $G = G(\Gamma, \rightarrow, \mu)$ over infinite number of processors, i.e. the available processor activities (p_{av}) of the multiprocessor system exceeds the required processor activity (p_{req}) of the given computation graph [9]. Since no approach is known to compute $l(T)'s$ when $p_{av} < p_{req}$, this method can be used as a heuristic to schedule computation graphs $G(\Gamma, \rightarrow, \mu)$ on an arbitrary number of processors.

In fact, the evaluation of $l(T)'s$ in the case when $p_{av} > p_{req}$ (unbounded resources), is completely independent of the task-processor assignments. Further, the optimum schedule in the case when $p_{av} < p_{req}$ (bounded resources) results from some specific partitioning of the given computation graph over the available processors and independent of a specific task-processor assignments.

Extending Graham's method to incorporate communication costs does not yield a realistic approximation of $l(T)'s$. This is because any evaluation of $l(T)'s$ should consider the topology factor of the multiprocessor system and thus the mapping of tasks to processors [2].

The approximate evaluation of the task level as the task completion time following a backward scheduling of the given computation graph over a multiprocessor system [6] [3], dose not estimate accurately the relative criticality among the tasks. In other words, the completion time of a given task $ct(T)$ that resulted from schedul-

[2]See section (3.2.1).

ing the reverse graph G_r dose not provide a good approximation of the task level $l(T)$ for the original graph.

We can distinguish between two factors that are mainly responsible for the inaccuracy inquired in using the $ct(T)$, resulting from the previous backward scheduling pass, to approximate the task level $l(T)$. Consider the case where a non-critical task got delayed due to the shortage of the number of available processors in the multiprocessor system, i.e. it could have started earlier if the multiprocessor system have enough processors. So the completion time of such a task following the current backward scheduling pass will be relatively high, which in turns will impose high relative priority for this task in the next forward scheduling pass. This is in contrast to its actual lower criticality value with respect to the original computation graph . We call this the *limited resources effect.*

Further, task completion times resulting from scheduling the reverse graph G_r on an **infinite number of processors**, are also not accurate approximation of task level values. This is mainly due to the fact that the used scheduling heuristic may serialize independent tasks (tasks that belong to independent paths in the given computation graph) on the same processor. We call this the *serialization effect* of the scheduling heuristic. The serialization effect is explained in the following example.

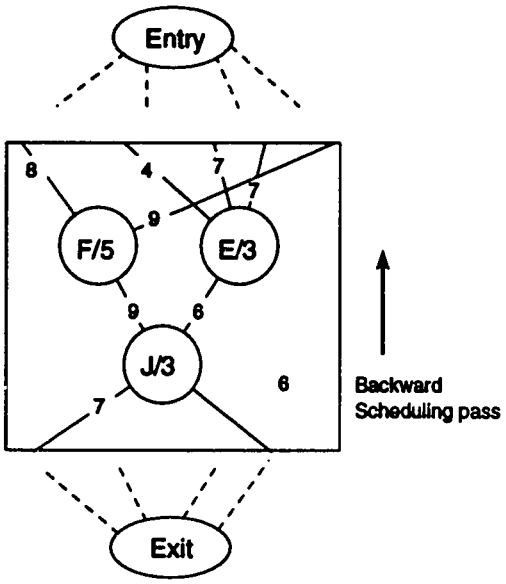

Figure 4.1: The serialization effect.

Example 1

Consider the sub-graph in Figure 4.1, and the following ETF sub-schedule over three processors:

$$est(F, p_1) = 14,\ est(F, p_2) = 23,\ est(F, p_3) = 23;$$

$$est(E, p_1) = 14,\ est(E, p_2) = 20,\ est(E, p_3) = 20;$$

$$eft(p_1) = 14,\ eft(p_2) = 17,\ eft(p_3) = 16;$$

where $eft(p)$ is the earliest time processor p becomes free. Assume that the current scheduling pass is a backward one, and the minimum starting time value among ready tasks at the current scheduling step is 14 and the second minimum value is 21. In this case ETF will break the tie between E and F since both of them have

the same earliest starting time. Let F be the selected task, then ETF will assign F on p_1 and does the following updates $ct(F) = 14 + 5$, and $eft(p_1) = 19$, and $est(E, p_1) = 19$. Consider the situation where the assignment of F dose not cause any other tasks to become newly ready, thus E will be the ready task with the least staring time value that is 19. Here, ETF will assign E to p_1 and does the following updates $ct(E) = 19+3$, and $eft(p_1) = 22$. Though E and F belong to different paths in the given computation graph, they have been serialized on the same processor. it is important to notice that even if we assume unbounded number of processors, p_1 will still be the processor that can reduce the starting time of E. Due to this serialization, the completion times of E and F do not maintain the actual relative criticality among them and with respect to the whole computation graph.

4.3 Longest Activity Path

Our proposed approach to approximate task level values combines some good scheduling heuristic for getting the task-processor assignments, along with a logical definition that accounts for both the limited resources effect and the serialization effect.

Definition 4.1 *Let T_i be a predecessor task of a given task T, i.e. $T_i \in pred(T)$. The latest time a message from T_i at $p(T_i)$ will reach $p(T)$ is given by $lmt_i = ct(T_i) + c(T_i, T).r(p(T_i), p(T))$. By considering all the predecessor tasks of T, the last time a message from the predecessors will reach T on $p(T)$ is defined as $lmt(T) =$*

$Max_{T_i \in pred(T)}[lmt_i]$, *and we call the predecessor task that satisfies this definition (the predecessor task that participated the latest message) as the dominant task with respect to* T.

Note that $lmt(T)$ specifies a lower bound on the earliest starting time of T on $p(T)$ according to the scheduling done so far to its predecessors. The effective earliest starting time $est(T, p(T))$ of a task T can differ form its last message time $lmt(T)$, since T might get delayed due to one of the effects mentioned above or both of them.

Definition 4.2 *Given a schedule* D, *the longest activity path* $(lap(T))$ *of a task* T *is defined as the largest sum of computations and communication costs along some path in* D *from the entry node up to the last message time* $lmt(T)$ *of* T.

Our approach to overcome the above two effects is to use the task earliest starting time to approximate its longest activity path. Now by considering only one predecessor task $T_i \in pred(T)$ of a task T, the longest activity path $lap(T)$ of T depends on the longest activity path $lap(T_i)$ and the earliest starting time $est(T_i, p(T_i))$ of the predecessor task T_i, and is equal to:

$$lap(T) = lap(T_i) + lmt(T) - est(T_i, p(T_i)). \tag{4.3}$$

Remember that we need to measure the longest activity along a path from the entry node to the last message time $lmt(T)$ of task T. By activity we mean the computations or communication costs. Note that $lap(T_i)$ measures the longest activity path from the entry node and up to the $lmt(T_i)$, so we need to measure the

activity time portion from $est(T_i, p(T_i))$ to $lmt(T)$ (since equation 4.3 has considered just one predecessor task T_i, so $lmt(T) = lmt_i$). In fact, the time interval from $est(T_i, p(T_i))$ to lmt_i is covered by the activity of T_i (the computation time of T_i and the communication requirement between T_i and T). If $lap(T_i) \neq est(T_i, p(T_i))$, then the time interval $[lap(T_i), est(T_i, p(T_i))]$ must have been caused by the serialization effect or the limited resources effect or both. Thus, the time interval from $lap(t_i)$ to the $est(T_i, p(T_i))$ must be removed from evaluation of $lap(T)$ since it represents an activity that dose not belong to the path to T through T_i.

By considering all the predecessor tasks of T, the $lap(T)$ is the longest activity path among all the paths to T through its predecessor tasks :

$$lap(T) = Max_{T_i \in pred(T)}\{lap(T_i) + lmt(T) - est(T_i, p(T_i))\}. \tag{4.4}$$

Since $lmt(T)$ is a constant value, we can rewrite 4.4 as:

$$lap(T) = lmt(T) + Max_{T_i \in pred(T)}\{lap(T_i) - est(T_i, p(T_i))\}. \tag{4.5}$$

In the case T is an entry node, i.e. $(pred(T) = \phi)$, then $lap(T) = 0$. So we can define the longest activity path for any task T as:

$$lap(T) = \begin{cases} 0 & \text{if } pred(T) = \phi \\ lmt(T) + Max_{T_i \in pred(T)}[lap(T_i) - est(T_i, p(T_i))] & \text{otherwise} \end{cases} \tag{4.6}$$

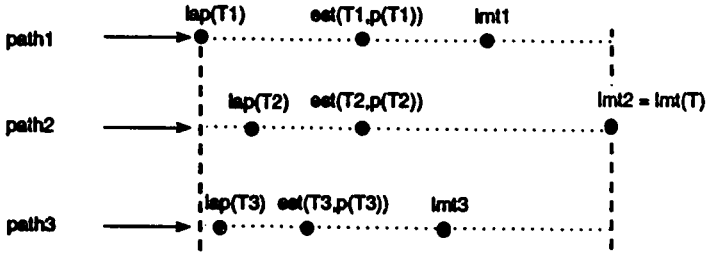

Figure 4.2: Longest activity path evaluation.

Note the definition of *lap* is a recursive one , i.e. to evaluate the $lap(T)$ of a task T, first we should have the *lap* values for all predecessor tasks of T $(lap(T') : T' \in pred(T)$. Actually the evaluation of the *lap* value for a given task T is performed in two consecutive steps. The first step finds the predecessor task $T_i \in pred(T)$ that maximize the value for $lap(T_i) - est(T_i, p(T_i))$ among all predecessors of T. The second step adds to $lap(T_i)$ the time interval $[est(T_i, p(T_i)), lmt(T))$ which is covered by the computations of some predecessors and the communication costs between some predecessors tasks and T.

example2 Given a task T, let T'' be the predecessor that has the smallest *lap* value, i.e. $lap(T'') = Min_{T' \in pred(T)}\{lap(T')\}$. One can realize that the recursive definition of $lap(T)$ aims to cover as much as possible of the time interval $[lap(T''), lmt(T))$ by the computations of some predecessors and the communication costs between some predecessor tasks and T. To understand this the reader is recommended to go through the following example.

Example 2

Consider the situation in figure 4.2 where the scheduler has just scheduled T that has three predecessors T_1, T_2, and T_3 which define three different paths to T. Their respective *lap* values are $lap(T_1), lap(T_2)$, and $lap(T_3)$, their earliest starting times are $est(T_1, p(T_1)), est(T_2, p(T_2))$, and $est(T_3, p(T_3))$, respectively. Note that T_2 is the dominant task of T, as the latest message time $lmt(T)$ of T is equal to lmt_2, i.e. $lmt(T) = lmt_2$. Our aim is to define $lap(T)$. Consider the following points:

1. the time intervals $[lmt_3, lmt(T))$ and $[lmt_1, lmt(T))$ are covered completely by the computation and communication of T_2,

2. along the three paths, the intervals $[est(T_i, p(T_i)), lmt_i)$ for: $i = 1, ..., 3$, are covered by the computation and communication cost of the respective T_i,

3. the intervals $[lap(T_i), est(T_i, p(T_i))$ for: $i = 1, ..., 3$, must be caused by one or both of the effects [3].

It should be clear that the path that has the smallest interval $[lap(T_i), est(T_i, p(T_i))]$ for $i = 1..3$, is the one contributing to the longest activity path; in this case it is the path along T_2.

The longest activity path value approximates the largest sum of computations and communication costs along a path in the given computation graph from the entry node till the latest message time of T which lower bounds the effective earliest

[3]The limited resources effects, and the serialization effect.

starting time of T. If we sum the task computation time $\mu(T)$ with its $lap(T)$ value following a backward scheduling pass, then this will approximate the remaining computations and communication costs from starting of T till the exit node in the current forward scheduling pass. We call this sum as the task effective remaining distance ($erd(T)$). In notations this can be written as $erd(T) = lap(T) + \mu(T)$. Note that backward scheduling pass is equivalent to scheduling the reverse graph G_r of the given graph problem $G = G(\Gamma, \rightarrow, \mu, c)$ on the multiprocessor system $S = S(P, R)$. The corresponding reverse graph G_r of the given computation graph G is obtained by reversing the all edge directions in G. Moreover, scheduling G_r starts with the exit node and continue upward till the entry node.

We have shown that the effective remaining distance gives a more refined task criticality measure than that of the task completion time. This is because the $erd(T)$ notion eliminates the effects of irrelevant activities from the evaluation of the task level and consider only the activities along an actual path in the computation graph from the entry node to the task T. A scheduling heuristic that incorporates the effective remaining distance as a criticality measure is introduced in the following section.

4.4 The CD/ERDETF Scheduling Heuristics

Scheduling computation a graph according to it task level values may result in increasing the processor idle time intervals upon the starting of the selected tasks, which in turns will increase the over all schedule finish time. This is because there is no guarantee that the ready task with the highest level value is always the one with the least starting time. Our proposed scheduling heuristic accounts for the idle time spent upon starting the task and is called Computation Driven / Effective-Remaining-Distance-Earliest-Task-First ($CD/ERDETF$). Ready tasks are selected according to highest value of

$$erd(T) - est(T),$$

where $est(T)$ and $erd(T)$ are the earliest starting time and the effective remaining distance of task T.

$CD/ERDETF$ is considered to compensate generally for the difficulties of global priority based scheduling heuristic with respect to processor utilization efficiency. To see this consider a task T having the highest priority among the set of ready to run tasks R. So we have:

$$erd(T) - est(T) \geq erd(T_i) - est(T_i) : T_i \in R,$$

which we can rewrite as:

$$erd(T) - erd(T_i) \geq est(T) - est(T_i) : T_i \in R.$$

Here we can distinguish between two cases. The first case is when T and T_i compete for different processors, i.e. $p(T) \neq p(T_i)$, then assigning T to start on processor $p(T)$ can not prevent T_i from being assigned on $p(T_i)$ later. The second case is when T and T_i compete for the same processor p, i.e. $p(T) = p(T_i) = p$, then either $est(T,p) \geq est(T_i,p)$, or $est(T,p) \leq est(T_i,p)$. If $est(T,p) \leq est(T_i,p)$ then we will have $erd(T) \geq erd(T_i) - \{est(T_i,p) - est(T,p)\}$, one can think of this as that $CD/ERDETF$ decreases the task level value of T_i by the extra amount of processor idle time that proceeds T_i if it is to be scheduled first. Applying the same argument for the case when $est(T,p) \geq est(T_i,p)$ then $erd(T) \geq erd(T_i)+[est(T,p)-est(T_i,p)]$. This can be visualized as a penalty for task T by increasing the priority value of T_i by the extra idle time that proceeds T.

When $CD/ERDETF$ lacks global information, like in the case when *erd's* values for all ready tasks are around each others, it switches to uniform scheduling with respect of tasks on processors, like ETF. Because this is the best strategy in the absence of information about task criticality values.

Reported results and empirical testing evaluations of scheduling heuristics proved that global scheduling heuristics with better processor idle time management are capable of producing shorter schedule finish times [6] [3]. In the case of $CD/ERDETF$, the decision function consists of two parts; $erd(T)$ which approximate the task criticality, and the *est* portion that is responsible of processor idle time optimization. Note that $est(T)$ is the dynamic part of the decision function, i.e. the earliest

starting time of a given tasks is evaluated while doing the scheduling, while $erd(T)$ portion of the decision function is a static one that is passed from the previous scheduling pass. Static decision functions can not reflect the actual situation of the current scheduling process, since they represents just approximation and this may not be accurate for the current scheduling process.

We are proposing to investigate more efficient combining of these two notions (erd and est) that can produce shorter finish times. So we will rewrite the scheduling decision function $d(T)$ of $CD/ERDETF$ as in the following:

$$d(T) = erd(T) - \kappa.est(T, p(T)),$$

where, κ is a positive weight. The effect of this weight is to modulate the processor idle time optimization.

Algorithm $CD/ERDETF$ uses two sets. Set R is used to store the current ready to run tasks, and set A is used to store tasks that have been already assigned to some processors. Functions $est(T,p), ct(T)$, and $eft(p)$ denote the earliest starting time of T on a processor p, the completion time of T, and the earliest time processor p becomes free, respectively. Further, the counter $\lambda_{pred}(T)$ is initialized to the number of predecessor tasks of T, and is decremented each time a predecessor task is assigned. So, a task T is considered ready whenever its $\lambda_{pred}(T) = 0$. $pred(T)$, and $succ(T)$ denotes the set of predecessor and successor tasks of a given task T. $CD/ERDETF$ takes a computation graph G, the multiprocessor system S, list of

task effective remaining distances , and the coefficient κ. It produces a schedule of G over S along with tasks effective remaining distances. In notation algorithm $CD/ERDETF$ is the following:

Algorithm CD / ERDETF.

Input: A computation graph $G = G(\Gamma, \rightarrow, \mu, C)$, a multiprocessor system $S = S(P, R)$, a list $L = \{erd(T) : T \in \Gamma\}$, a value for κ.

Output: The resulted schedule $\{(est(T), p(T)) : T \in \Gamma\}$, and the list $L = \{(erd(T) = lap(T) + \mu(T) : T \in \Gamma\}$.

(1) Initialization: $R = \{T : T \in \Gamma$ and $pred(T) = \phi\}$; $A = \phi$; $est(T, p) = 0$ for each $T \in R$ and each $p \in P$; $lap(T) = 0$ for each $T \in R$; $eft(p) = 0$ for each $p \in P$, intialize $\{\lambda_{pred}(T) : T \in \Gamma\}$

(2) While $|A| < n$ Do

Begin

(2.1) /* Find the ready task $T^* \in R$ and processor $p^* \in P$ that maximize the scheduling decision function */

$erd(T^*) - \kappa.est(T^*, p^*) = Max\{erd(T) - \kappa.est(T, p) : T \in R$ and $p :\in P\}$

(2.2) /* Schedule T^* on p^* and do the required updates*/

Assign T^* on p^*; $ct(T^*) = est(T^*, p^*) + \mu(T^*)$; $eft(p) = ct(T^*)$; $R = R - \{T^*\}$; $A = A \cup \{T^*\}$;

For each $T \in R$ update it's $est(T, p^*)$ to be

$$est(T, p^*) = Max\{est(T, p^*), eft(p^*)\}.$$

(2.3) /* Compute the $lap(T^*)$ value of the Scheduled task T^* */ $lmt(T^*) = Max_{T' \in pred(T^*)}\{ct(T') + c(T', T^*).r(p(T'), p^*)\}$;

Find $T \in pred(T^*)$ so that

$$lap(T) - est(T, p(T)) = Max_{T' \in pred(T^*)}\{lap(T') - est(T', p(T'))\};$$

Use T to update the lap value for T^*:

$$lap(T^*) = lap(T) + lmt(T^*) - est(T, p(T)).$$

(2.4) /* Check if any of T^*'s successor tasks becomes ready,

and add it to R */

For each task $T \in succ(T^*)$

$\lambda_{pred}(T) = \lambda_{pred}(T) - 1$;

If $\lambda_{pred}(T) = 0$ Then

$R = R \cup \{T\}$: T becomes newly ready to run;

Evaluate $est(T, p)$ for each $p \in P$ as

$$est(T, p) = Max\{Max_{T' \in pred(T)}\{ct(T') + c(T', T).r(p(T').p)\}, eft(p)\}$$

End.

The strategy of $CD/ERDETF$ is to schedule tasks in sequence along an immediate chain of tasks until the task decision function value of some task in the chain drops below the decision value of another ready task. Then $CD/ERDETF$ shifts to a the next critical chain of tasks. In fact, $CD/ERDETF$ tries at each scheduling step to schedule the task that is currently heading the longest path to the exit node. The decision function of $CD/ERDETF$ enforces that a task is assigned to a processor that reduces its starting time. So, tasks with substantially high level values, belonging to the same chain, are most probably assigned to the same cluster of processors that is able to reduce their starting times

Note that the strategy of $CD/ERDETF$ algorithm is to use *erd's* resulting from scheduling the reverse graph G_r on the multiprocessor system $S(P, R)$, as a quantifier of task priority in scheduling G. We expect *erd*'s to give more refined approximations of task levels that statistically minimize a schedule fish time. Therefore we can extend this approach to alternatively schedule G_r and G over system S and passing the resulted task *erd* from one scheduling pass to the next scheduling pass in order to search in a deterministic manner in the space solution. The initial step of the iterative refinement process could use the $cT's$ following a backward scheduling pass, using the best known local heuristic (ETF), as task criticality measure for the next forward scheduling using $CD/ERDETF$.

4.5 Analysis of the CD/ERDETF

In this section we will explain the main characteristics of $CD/ERDETF$ along with formal definition of the behavior of its scheduling decision function. Its time complexity will be established in the following theorem.

Theorem 4.1 *Given a multiprocessor system $S = S(P, R)$ with p processors, and a precedence computation graph $G = G(\Gamma, \rightarrow, \mu, C)$ with n tasks, the time complexity of $CD/ERDETF$ is of order $O(pn^2)$.*

Proof. The outer loop will execute n times to assign the n tasks in the given computation graph. This is due to statement 2.2 that will schedule one task at each iteration. Statement 2.1 executes $n.p$ times to find task T^* among at most n tasks in R, and processor p^* among p processors, so it is of order $O(np)$. To update the new $est's$ for at most n tasks in R, statement 2.2 needs to execute n times, so it is of order $O(n)$. Statement 2.3 needs to make one pass over the set $pred(T^*)$ in order to find $lmt(T^*)$ and the predecessor T of task $T*$, this is of order $O(n)$. The condition $\lambda_{pred}(T) = 0$ occurs once for each task, and statements 2.4 executes in total n^2 times. So the overall time complexity of the algorithm is $O(pn^2)$ •

Let $d(T)$ denote the value of the scheduling decision function for the task T, so $d(T) = erd(T) - \kappa.est(T, p(T))$. Further let $do(T)$ denote the scheduling decision order for T. So, if the given computation graph G has n tasks then $1 \leq do(T) \leq n$ is true for all tasks in G.

Lemma 4.1 *ERDETF uses a scheduling decision function that is a non-increasing function along any path* $J : (T_{J1} \rightarrow T_{J2} \rightarrow ... \rightarrow T_{Jm})$ *in* G *or* G_r.

Proof. Given two immediate tasks, T and T' such that $T' \in pred(T)$ we want to prove that $d(T') > d(T)$, i.e. $erd_b(T') - \kappa.est_f(T', p(T')) \geq erd_b(T) - \kappa.est_f(T, p(T))$.

The subscripts for erd_b and est_f denotes that the current scheduling pass is forward (est_f), and the used effective remaining distance values have resulted from the previous backward scheduling pass (erd_b). Note that T is a predecessor task for T' in G_r.

First we need to establish the relation among $erd_b(T')$ and $erd_b(T)$. In fact, we have two cases:

1. the directed path along T is the one that contributed to the evaluation of the longest activity path of T' in G_r, so T is the predecessor task that satisfies $Max_{T'' \in pred(T')}\{lap(T'') - est_b(T'', p(T''))\}$ in G_r, and we can say the following:

   $$erd_b(T') = lap(T) + lmt(T') - est_b(T, p(T)) + \mu(T'),$$

   and by definition $\quad erd_b(T) = lap(T) + \mu(T)$.

   Since $lmt(T') \geq est_b(T, p(T)) + \mu(T)$ is true for any predecessor task $T \in pred(T')$ in G_r, then $erd(T')$ is at least greater than the sum $lap(T) + \mu(T)$. Thus $erd(T') > erd(T)$.

2. the path through some other predecessor T_1 is the one that contributed to the evaluation of the longest activity path of T' in G_r. Let T_1 be the predecessor

that satisfies $Max_{T'' \in pred(T')}\{lap(T'') - est_b(T'', p(T''))\}$ in G_r, thus:

$$lap(T_1) - est_b(T_1, p(T_1)) \geq lap(T) - est_b(T, p(T)) \Rightarrow$$

$$lap(T_1) - est_b(T_1, p(T_1)) + est_b(T, p(T)) \geq lap(T) \Rightarrow$$

$$lap(T_1) - est_b(T_1, p(T_1)) + est_b(T, p(T)) + \mu(T) \geq lap(T) + \mu(T) \Rightarrow$$

$$lap(T_1) - est_b(T_1, p(T_1)) + est_b(T, p(T)) + \mu(T) \geq erd_b(T),$$

again since $lmt(T') \geq est_b(T, p(T)) + \mu(T)$ is true for any predecessor task $T \in pred(T')$ in G_r, then

$$lap(T_1) - est_b(T_1, p(T_1)) + lmt(T') \geq erd_b(T) \Rightarrow$$

$$lap(T_1) - est_b(T_1, p(T_1)) + lmt(T') + \mu(T_1) > erd_b(T)$$

note that the left hand side of the above inequality is equal to $erd(T')$ itself, thus $erd(T') > erd(T)$.

Further we need to establish the relation between $est_f(T', p(T'))$ and $est_f(T, p(T))$. As $T' \in pred(T)$ in G and $\mu(T) > 0$, then $est(T', p(T')) < est(T, p(T))$ is true for all successor tasks of T'.

As $erd_b(T') > erd_b(T)$ and $est_f(T', p(T')) < est_f(T, p(t))$, always maintained, $erd_b(T') - \kappa.est_f(T', p(T')) \geq erd_b(T) - \kappa.est_f(T, p(T))$ is satisfied for all immediate tasks of T'. It is obvious that the above reasoning applies also to directed paths in G_r •

Lemma 4.2 *CD/ERDETF uses a scheduling decision function that is a non-increasing function along any increasing sequence of decision orders.*

Proof. We want to proof that $d(T) \geq d(T')$ is always true whenever $do(T) < do(T')$. Consider the case when T has been scheduled at some decision order $do(T)$, then the $d(T)$ must satisfy $Max_{T' \in R}\{d(T')\}$ for any ready task T'. Further we have $d(T') > d(T'')$ for all $T'' \in succ(T')$ see Lemma 4.1. Note that the decision order along any chain of immediate tasks in G must be increasing [4]. Therefore, the decision function is a non-increasing function along any increasing sequence of decision orders $d(T) \geq d(T') \geq d(T'')$ whenever $do(T) < do(T')$ and T'' is some successor of any unscheduled but ready to run task •

Theorem 4.2 *CD/ERDETF does not satisfy the worst case bound of Hwang processor-driven/earliest-task-first (ETF).*

Proof. Hwang proved that the length of an ETF schedule w_{ETF} is upper bounded by the sum of Graham's bound of list scheduling and the communication requirements. In notation this can expressed as:

$$W_{ETF} \leq (2 - 1/p)w_{opt} + C$$

where w_{opt} is the finish time of the optimum schedule when communication between tasks is not considered, C is the maximum pessimistic communication requirement

[4] A task is scheduled before its successor.

of all chains in the given computation graph, and n is the number of processor in the system. ETF schedules tasks so that the starting times of successively scheduled tasks forms a non-increasing sequence in time. In fact, the bound is based on the fact that given any ETF schedule D_{ETF} with finish time w_{ETF}, it is always possible to construct a list of immediate tasks in D_{ETF} that covers all the time points in the interval $[0, w_{ETF})$ where at least one processor is idle.

Given an $CD/ERDETF$ schedule D with finish time w, the set of all points in the time interval $[0, w)$ can be partitioned into two subset A and B. A constitutes the set of all time points when all the processor are busy (executing some tasks), and the set B defines all the points in time when at least one processor is idle. Note that A and B are the disjoint union of open intervals. Assume B is the disjoint union of q open intervals ,$B = \bigcup_{1 \leq i \leq q}(b_{li}, b_{ri})$, such that $b_{li} < b_{ri}$. We need to prove that it is not possible always to construct a chain of immediate tasks $X : T_{x1}, \rightarrow, T_{x2}, \rightarrow, ..., \rightarrow, T_{xk}$ in D so that T_{xk} completes at w, i.e. ($ct(T_{xk} = w$, and X covers B. X covers B means that the total length of B in D is not longer than the sum of computations and the pessimistic communication costs along X.

for the proof we consider the following counter example where a tasks T has been scheduled to some processor $p(T)$ with starting time $est(T)$ such that $est(T) \in B$ and is not a boundary point. By the definition of B there must be some processor p_ϵ and a positive integer ϵ so that p_ϵ is idle in the interval $(est(T) - \epsilon, est(T))$. Consider the case where the last message time of T on P_ϵ is less than $est(T)$, i.e.

$lmt(T, p_\epsilon) < est(T)$. Let T' be another task that is independent of T [5]. and has been scheduled to p_ϵ with $est(T', p_\epsilon) > est(T, p(T))$. The natural question which comes to the mind is why T has not been started until $est(T)$ while P_ϵ was idle before and during this time. One possible answer is that $do(T') > do(T)$ and thus $d(T') > d(T)$ (see Lemma 4.1). This means that though $est(T', p_\epsilon) > est(T, p_\epsilon)$, T' has been scheduled on p_ϵ before scheduling T on $p(T)$. So an $CD/ERDETF$ schedule may leave intervals, such as $(est(T, p_\epsilon), est(T.p(T)))$, which are not covered by any chain of computations and communications •

4.6 Iterative Refinement

The basic idea of $CD/ERDETF$ is to use an initial estimate of the the task level obtained by scheduling the reverse graph G_r on the multiprocessor system $S(P, R)$ using the local scheduling heuristic ETF. The logical extension of this backward-forward scheduling process is to alternatively schedule G_r and G on the multiprocessor system S while passing the task *erd* values from one scheduling pass to another. The objective of this iterative process is to search a solution with shorter finish times as a result of using a more accurate estimate of task level values out of successive refinements.

Local scheduling heuristics try to minimize the resulting schedule overall finishing

[5]Neither of them is a predecessor of the other.

time by locally minimizing the processor idle time. Let H_l stands for the local scheduling heuristic PD/ETF that will be applied to schedule the computation graph G_r on the multiprocessor system $S(P,R)$. The scheduling process can be denoted by:

$$H_l(G_r, S(P,R)) = \{(est(T_i), p(T_i)) : T_i \in \Gamma\},$$

such that H_l takes two input parameters: the corresponding reverse graph G_r of the original computation graph and the multiprocessor system, and returns the resulting schedule as its output. Further, let H_g denote the global scheduling heuristic $CD/ERDETF$, again we can denote this scheduling process by:

$$H_g(G, S(P,R), L) = \{(est(T_i), p(T_i)) : T_i \in \Gamma\} \; and \; \{erd(T) : T_i \in \Gamma\}.$$

where L is the list of task erd values. $CD/ERFDETF$ actually use a non-increasing function of task level values as their task selection strategies. So increasing the level of a task, means giving it more priority to start earlier on some idle processor.

Actually the Iterative Refinement Scheduling (IRS) consists mainly of the following steps

1) Initialization: schedule G_r on $S(P,R)$ using

$H_l(G_r, S(P,R)) = \{(est(T_i), p(T_i)) : T_i \in \Gamma\}$.

1.1) Obtain the set of task completion times $L = \{ct(T_i) : T_i \in \Gamma\}$.

1.2) Set iteration counter $i = 1$.

2) Repeat

2.1) Perform forward scheduling of G on $S(P,R)$ using

$H_g(G, S(P,R), L) = \{(est(T_i), p(T_i)) : T_i \in \Gamma\}$ and $\{erd(T_i) : T_i \in \Gamma\}$.

2.1.1) Update the list $L = \{erd(T_i) : T_i \in \Gamma\}$

2.1.2) Obtain the forward schedule overall finishing time

$\omega_f = Max\{ct(T_i) : T_i \in \Gamma\}$.

2.2) Perform backward scheduling of G on $S(P,R)$ using

$H_g(G_r, S(P,R), L) = \{(est(T_i), p(T_i)) : T_i \in \Gamma\}$ and $L = \{erd(T_i) : T_i \in \Gamma\}$.

2.2.1) Update the list $L = \{erd(T_i) : T_i \in \Gamma\}$.

2.2.2) Obtain the backward schedule overall finishing time

$\omega_b = Max\{ct(T_i) : T_i \in \Gamma\}$.

2.3) Find the best ω at the current iteration $\omega[i] = Min\{\omega_f, \omega_b\}$, and increment the iteration counter $i = i + 1$.

Until $(i > Max)$ or (ω converges) or (ω oscillates)

Each iteration requires first scheduling G_r, then scheduling G. In fact, the concept of utilizing the $(erd(T))$ obtained from the previous scheduling pass as an indicator of task criticality in the current scheduling pass, incorporates the effects of computations, communications, and network latency along an arbitrary chain of

tasks. In the following task level value and task effective remaining distance will be used interchangeably.

IRS tries to investigate more accurate approximation of task level values. In fact, the decision function of a global scheduling approach is affected at each step by the following two factors:

- the so far discovered path, in the current scheduling pass, that T lies on, and
- the $l(T)$ value obtained from the previous scheduling pass.

These two factors are so related between any two consecutive scheduling passes.

The first factor imposes explicitly the effect of the so far discovered path that T lies on in the current scheduling pass, on enhancing the approximate evaluation of task level values obtained for the next scheduling pass. Further, it combines, in the current scheduling pass the effects of both precedence constraints and the so far task-processor assignments, and thus determines when a new task can be considered as ready.

However the second factor by it self results in blindly increasing the task level values, in the current scheduling pass, for tasks with small level values in the previous scheduling pass and vice versa. In fact, the effect of this factor is dominating, because obtained *red's* values control the decision functions of global scheduling heuristics. This is especially at the begging of the scheduling process as the task *erd* values are much larger than the corresponding task *est* values

In fact, the iterative refinement optimization technique is not based on increasing the task level values in the next scheduling pass for tasks with small task level values in the current scheduling pass. However, it is based on trying to increase the task level values in the next scheduling pass for more critical tasks whose delay in the current pass causes an increase in the overall schedule finishing time. Actually, it is very crucial for IRS not to increase task level values for noncritical tasks.

Chapter 5

Performance Evaluation and Analysis

In this chapter we present a performance comparison for PD/ETF, $CD/HLETF^*$, and $CD/ERDETF$. The performance comparison of $HLETF^*$, and $ERDETF$ is considered with in the iterative refinement approach. We considered a large set of randomly generated task-graphs as the workload for testing the relative performance of the algorithms.

The iterative refinement process is an optimization technique for global scheduling heuristics. The cost for IRS is low and imposes a linear effect on the time complexity of the heuristic it is applied on. IRS relies on using a more refined estimations of task levels through out the iterative process. In our empirical testing we limited the number of iterations to 100.

5.1 Workload Generation and Empirical Testing

To evaluate the performance of $CD/ERDETF$, a random graph generator RGG is used to generate precedence graph-problems with communication costs. RGG allows the generation of computation graphs with various properties. The graph size ranges from about 185 to 218, and the task computation time ranges from 3 to 17 time units. The following parameters control the average number of levels, the average communication cost per edge, and the number of processors and topology.

1. The ratio of the average communication cost of each edge (C_{ave}) to the average computation time of each task (μ_{ave}). This is denoted by $\alpha = C_{ave}/\mu_{ave}$. Seven values for α in the range 0 to 3 by steps of 0.5 are considered.

2. The parallelism degree of the given computation graph (β), i.e. the average number of tasks in the set of ready to run tasks over the number of processors. An approximation of β can be formulated as the concurrency degree of the graph over the number of available processors in the system. The concurrency level of a given computation graph is defined as the average number of tasks per level (N_t/N_l). So β can be defined as $N_t/(N_l.p)$, where p is the number of processors in the system. We considered seven parallelism degrees, these are $(0.5, 1, 2, 2.5, 3, 4, 5)$.

3. Two processor topologies are considered in the testing process. These are Fully-Connected (FC) and Hypercube (HC). The number of processor in

each topology is 8. Processor topology factor is denoted by ρ.

For each problem instance, the three heuristics PD/ETF, $CD/HLETF^*$, and $CD/ERDETF$ are used to schedule 30 graph problems. The shortest schedule finish time (w_{best}) generated by any of the heuristics for each computation graph is used as a reference for the comparison (the best solution we know). To compare the schedule finish time (w_h) of some heuristic H, for a given graph problem, to the reference w_{best}, we use this formula$((w_h - w_{best})/w_{best}) \times 100$ that represents the percentage deviation of w_h from w_{best}. The average percent deviation of each heuristic with respect to w_{best} at each problem instance (β, α, ρ) is reported. Separate plots are prepared for each topology. Also the average least number of iterations ($ALNI$) that generate the heuristic best performance at each problem instance is also reported in a separate plot. For each heuristic there will be two plots to report the finish time average percent deviation for the FC and HC. Another similar two plots are generated to report the $ALNI$. The legend **par** in the following figures specifies the parallelism degree of the corresponding curve.

5.2 Analysis of PD/ETF

For PD/ETF, increasing the parallelism degree in the computation graph, results in a better performance. This is because the strategy of PD/ETF is to reduce locally the processor idle times. Therefore the larger number of tasks competing for

the same processor at higher parallelism degrees gives the heuristic the opportunity to locally maximize the processor efficiency. In fact, ETF relies on overlapping computation with communication. Figure 5.1 shows this effect for the FC topology where the least average percent deviation (APD) for PD/ETF is obtained when β is 5. However, for low to moderate parallelism $0.5 \leq \beta \leq 4$, PD/ETF deviates significantly from w_{best} by about (16%) on the FC topology, and 17% on the HC topology. PD/ETF performs well only for computation graphs with high parallelism degree provided that the communication requirements and the network latency are moderate. Figure 5.2 shows that increasing α of the given computation graph yields a poor performance on the HC topology, about (15 %) deviation, even at high β. This is because changing the topology from FC to HC has the effect of increasing the communication requirements on the original graph, since assigning a task T to a processor p implicitly affects the overall finish time due to the connectivity of p and its associated communication costs.

5.3 Analysis of $CD/HLETF^*$

Figures 5.3 and 5.4 shows that the worst performance of $CD/HLETF^*$ on the FC topology is 5% deviation at $\beta = 3$, while at higher parallelism degree $\beta = 5$, $CD/HLETF^*$ performed better. Further, the worst performance of $CD/HLETF^*$ for the HC is 5% deviation at- $\beta = 5$. Figures 5.5 and 5.6 shows that the average

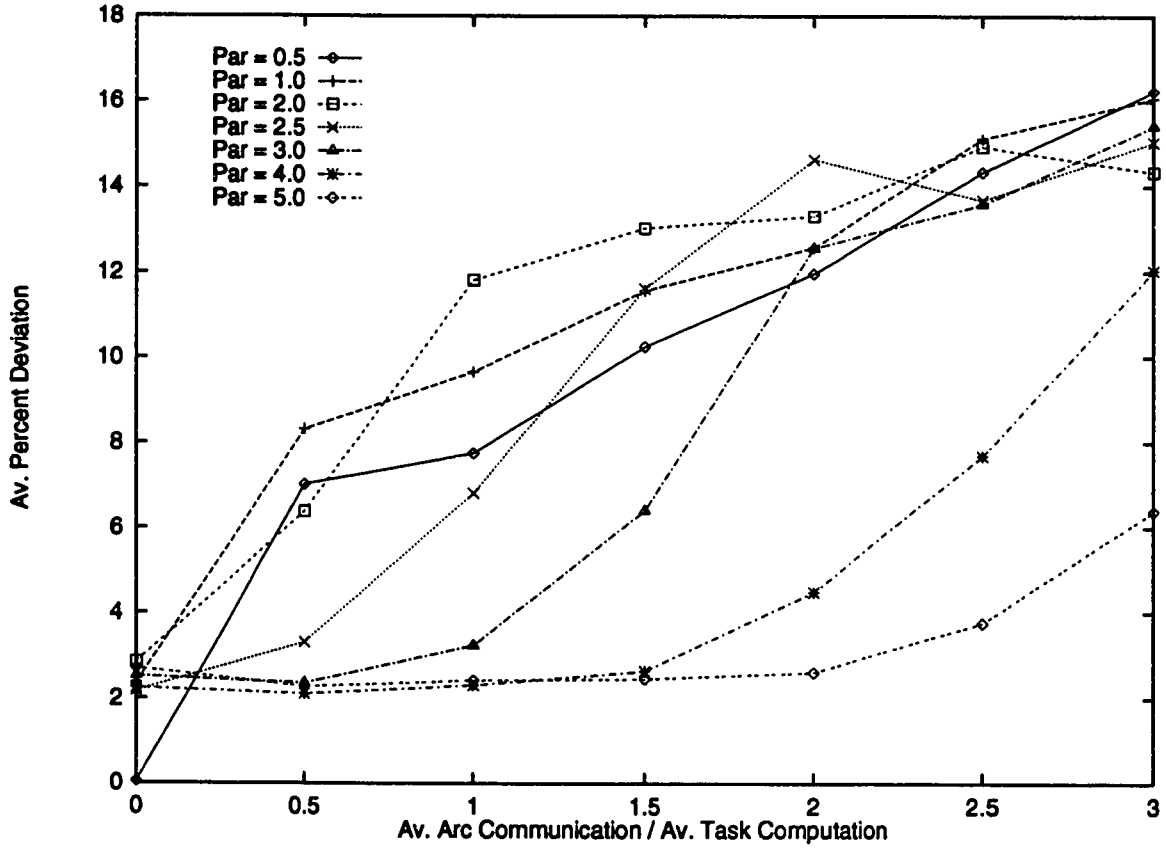

Figure 5.1: The relative performance of PD/ETF for FC Topology

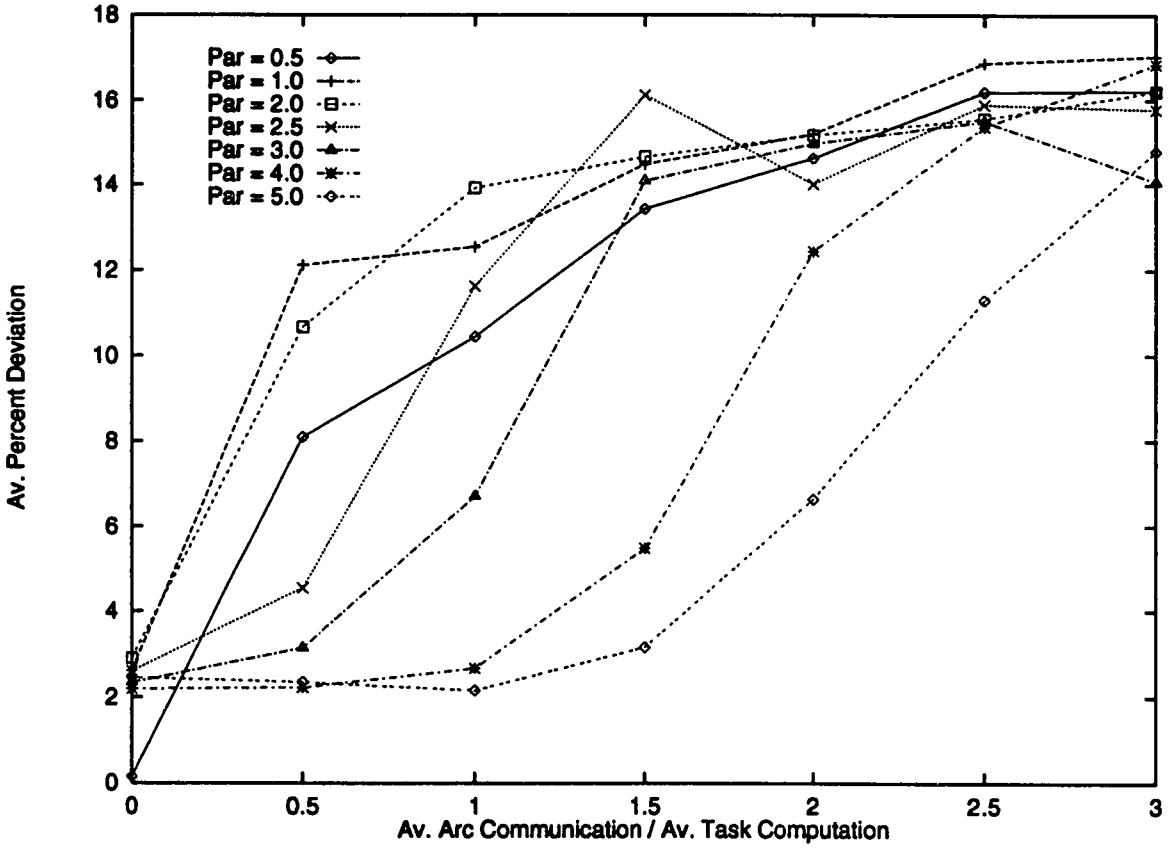

Figure 5.2: The relative performance of PD/ETF for HC Topology

number of iterations $CD/HLETF^*$ required to reach its best performance depends on the value of α and to a lesser extent on the value of β. In fact, fixing β at a specific value and stepping over α values, increases the number of iterations. This is because increasing the average value of the communication edges increases the irregularity in the graph problem , which in turns requires more iterations to explore various possibilities.

5.4 Analysis of $CD/ERDETF^*$

The objective here is to compare the scheduling heuristic $CD/ERDETF$ to both PD/ETF and $CD/HLETF^*$, by stepping over the granularity levels, parallelism degrees, and network latencies. Two versions of $CD/ERDETF$ are used. One with low κ value ($\kappa = 1$), and the other is with high κ value ($\kappa = 100$).

5.4.1 The effects of β and α

Figures 5.7 to 5.10 show that the best results are obtained form the two versions of $CD/ERDETF$. $CD/ERDETF$ outperformed both PD/ETF and $CD/HLETF^*$ for all studied granularity levels and parallelism degrees. The best performance of $CD/ERDETF$ over PD/ETF on FC and HC are about 15% and 16% respectively. The best performance of $CD/ERDETF$ over $CD/HLETF^*$ is the same for both FC and HC, and it is about 5%.

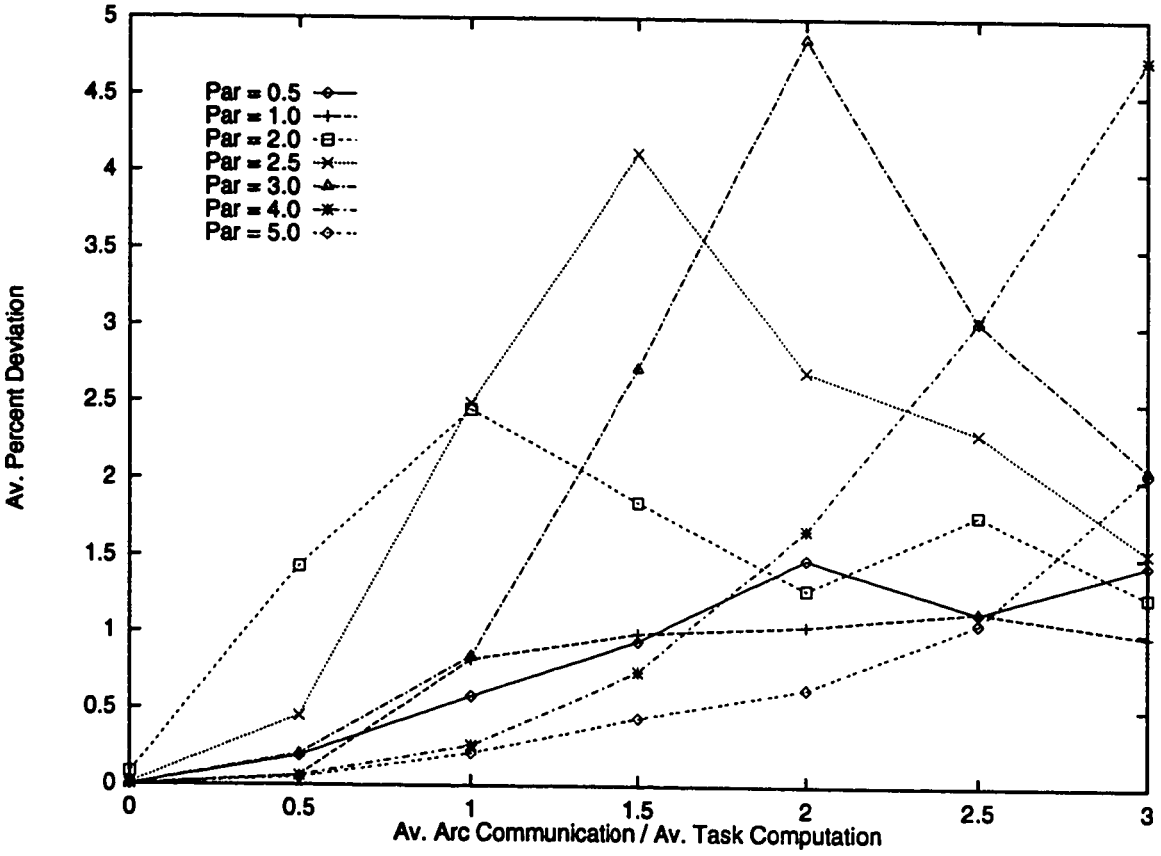

Figure 5.3: The relative performance of $CD/HLETF^*$ for FC Topology

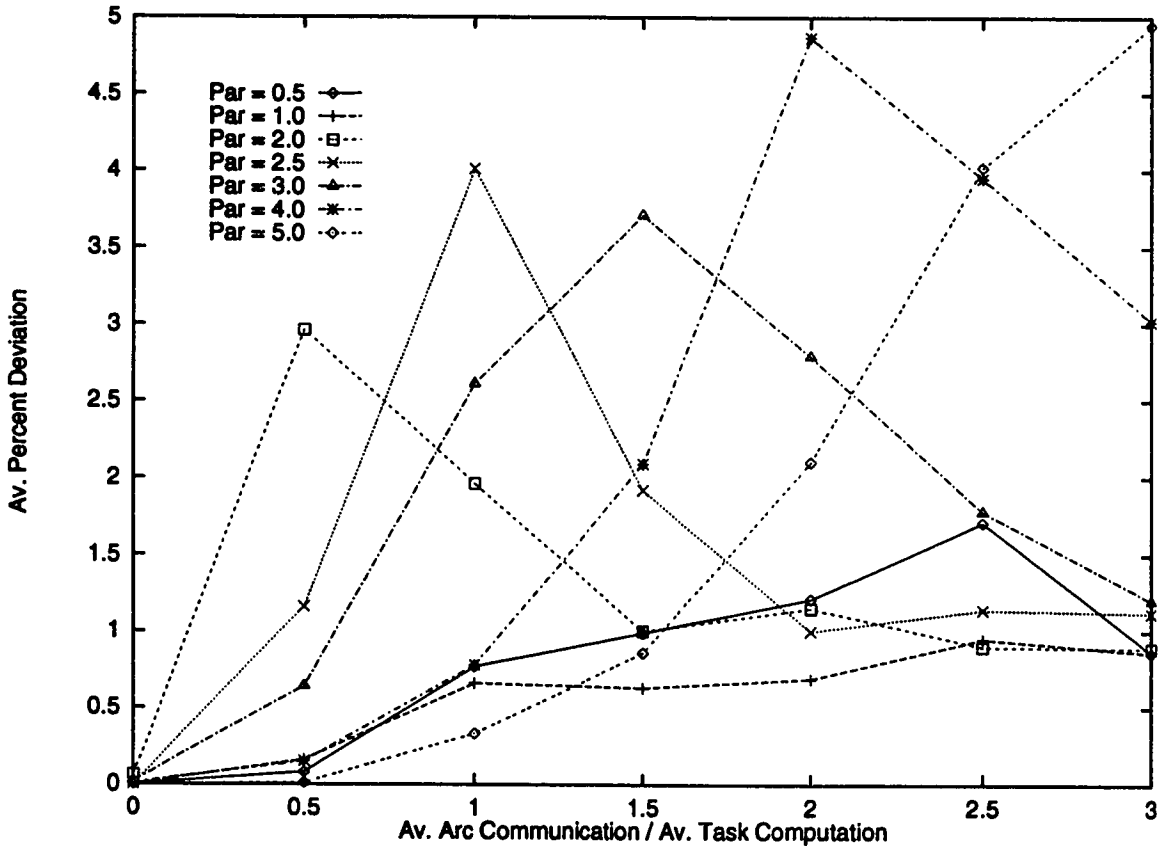

Figure 5.4: The relative performance of $CD/HLETF^*$ for HC Topology

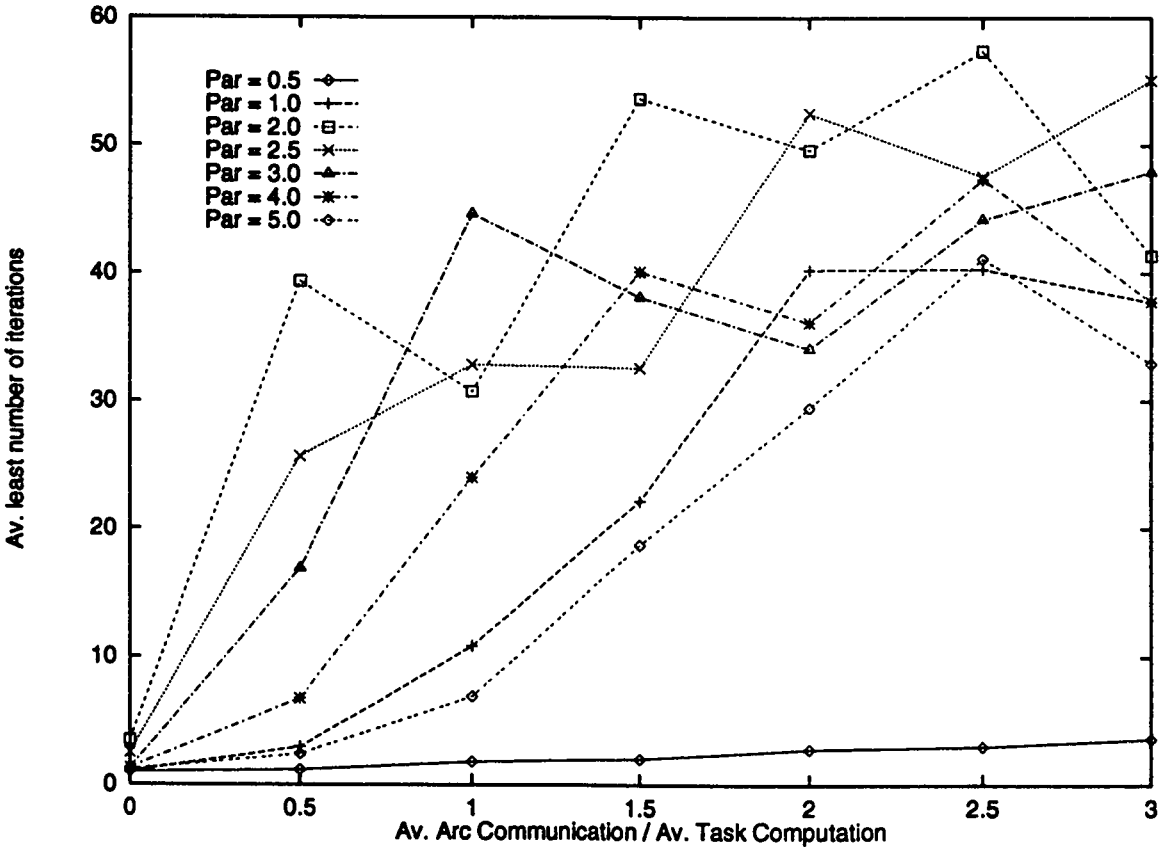

Figure 5.5: ALNI of $CD/HLETF^*$ for FC Topology

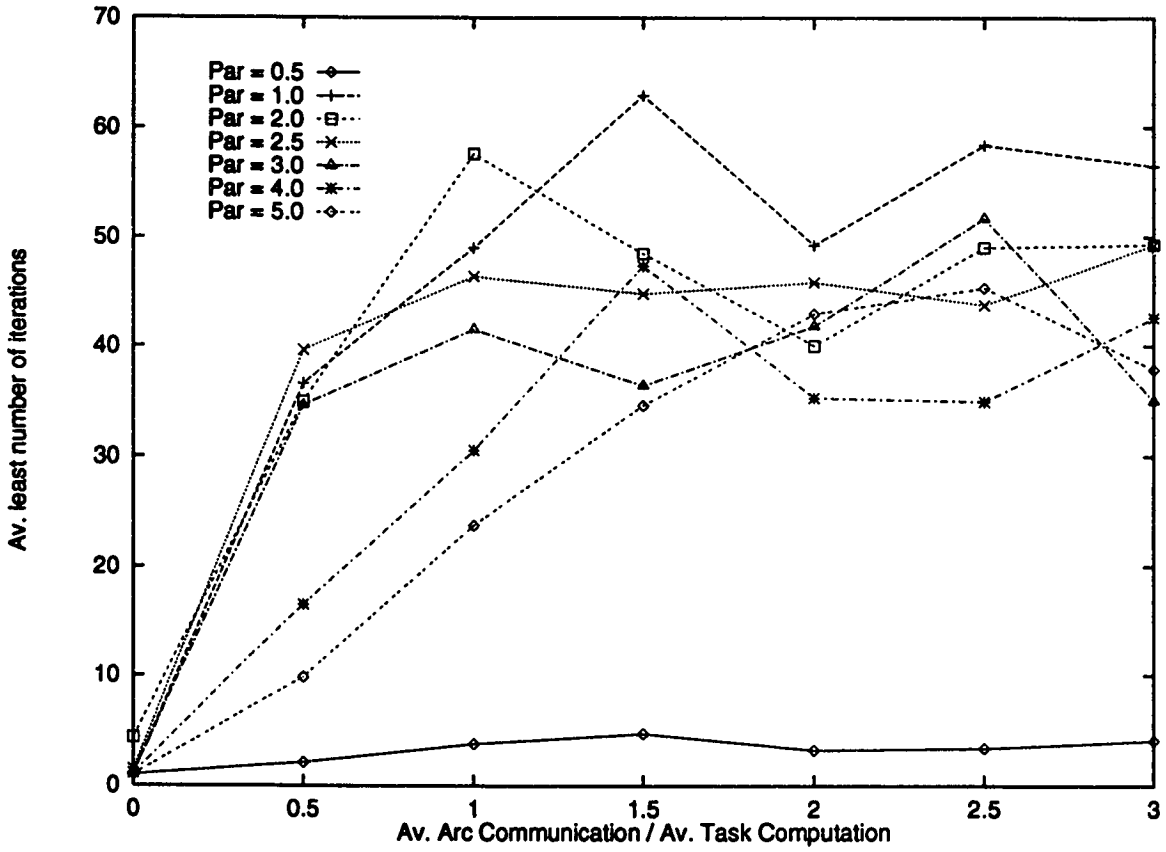

Figure 5.6: ALNI of $CD/HLETF^*$ for HC Topology

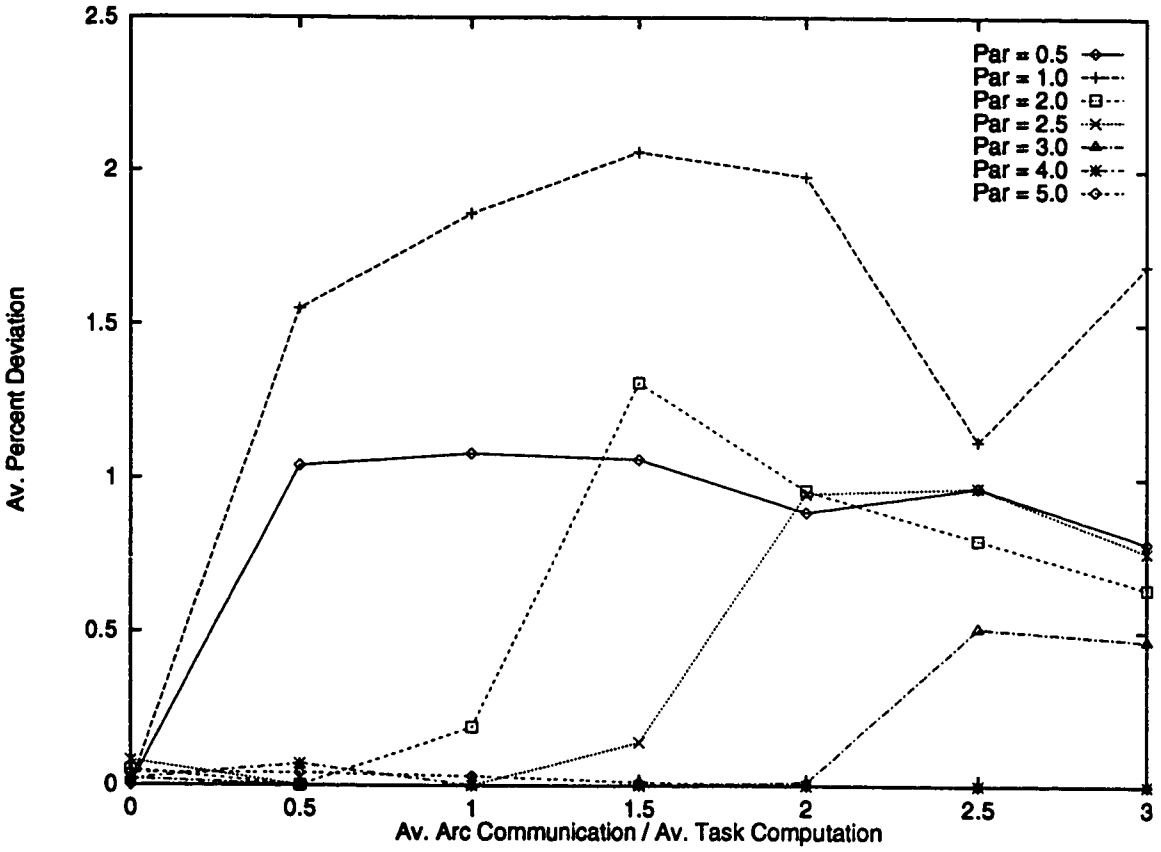

Figure 5.7: The relative performance of CD/ERDETF for FC Topology ($\kappa = 100$)

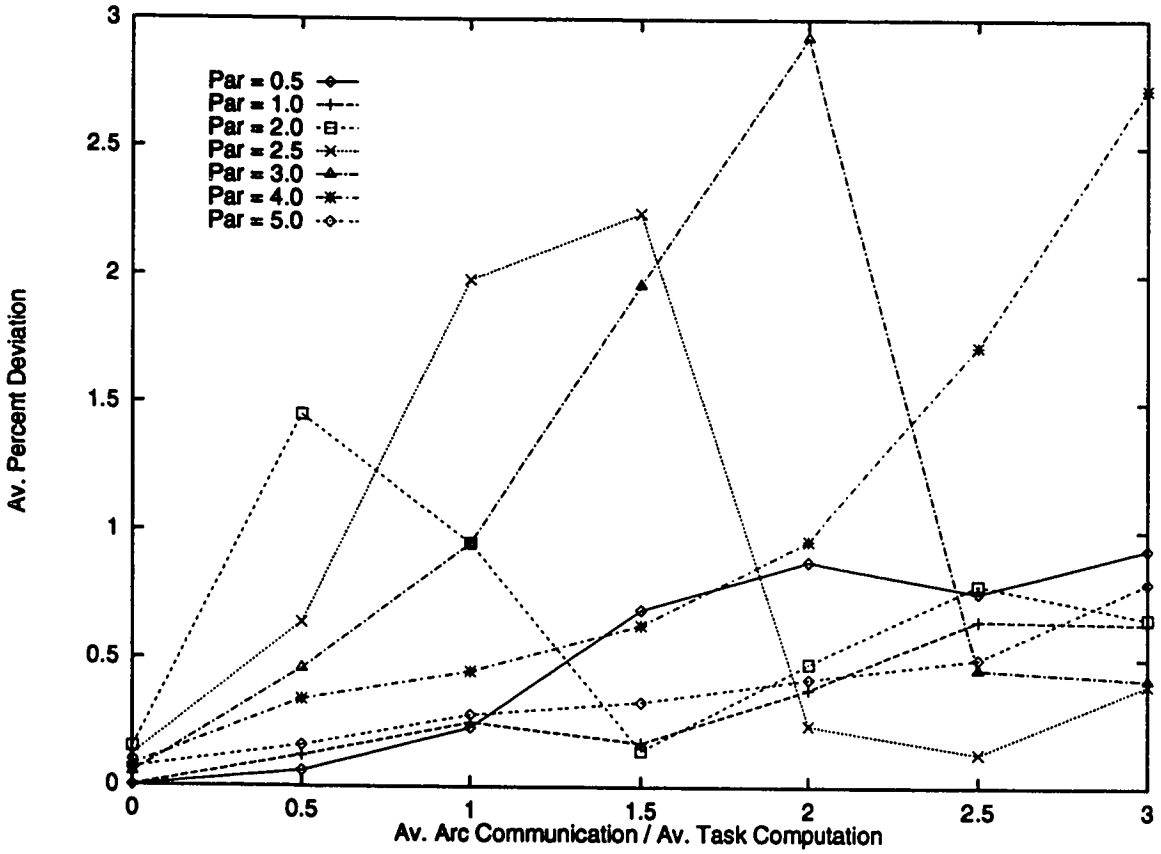

Figure 5.8: The relative performance of CD/ERDETF for FC Topology ($\kappa = 1$)

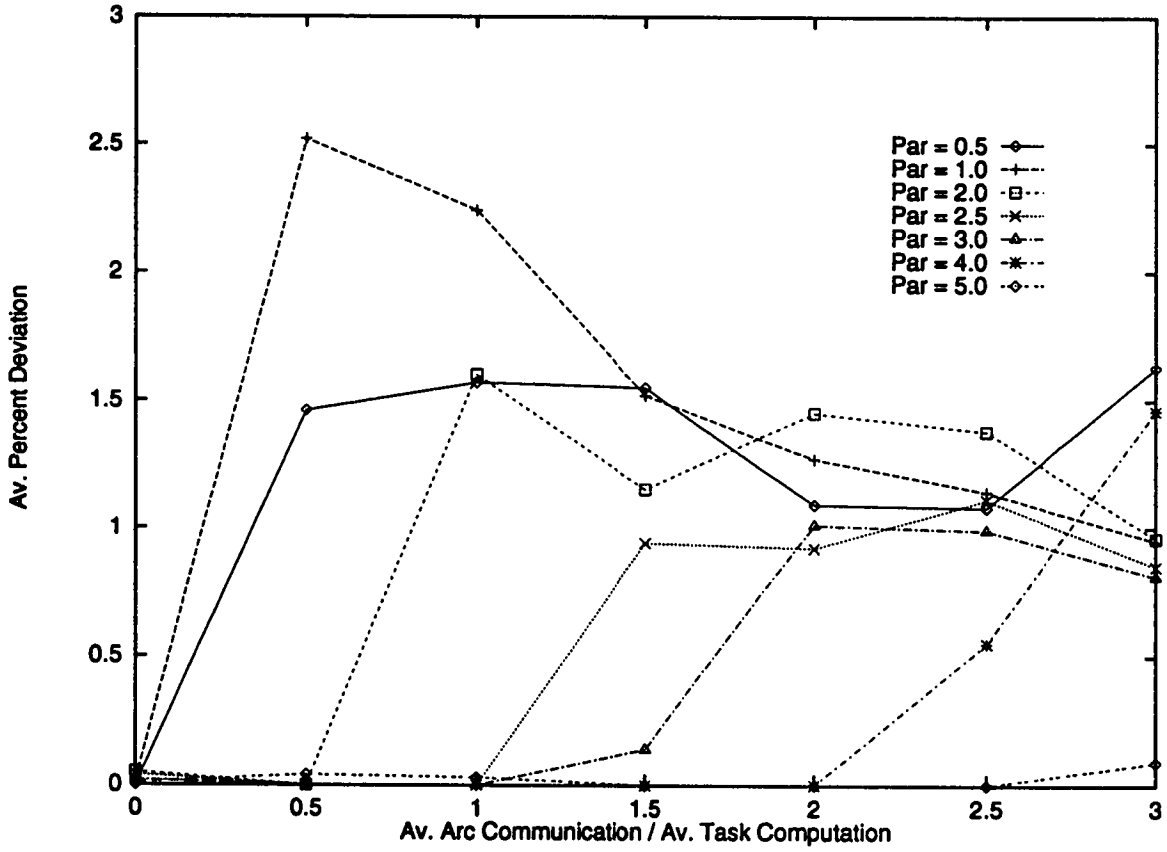

Figure 5.9: The relative performance of CD/ERDETF for HC Topology ($\kappa = 100$)

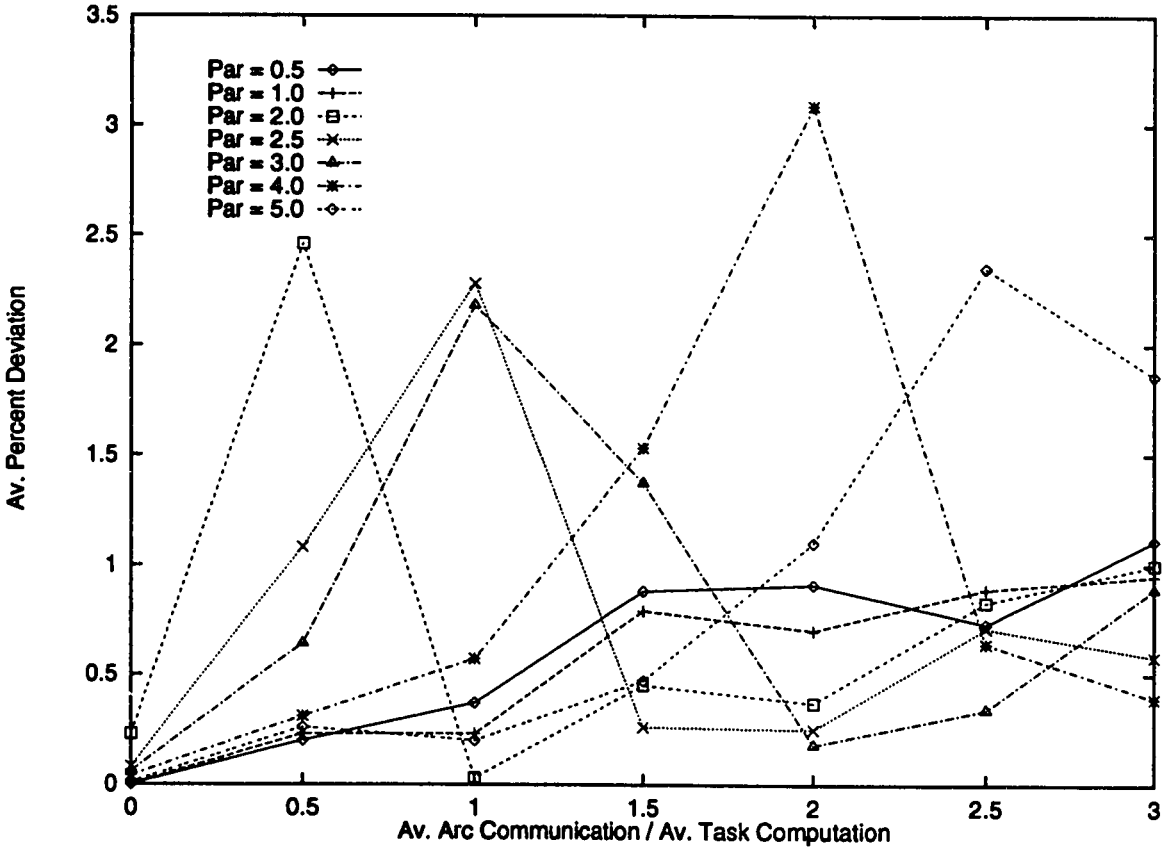

Figure 5.10: The relative performance of CD/ERDETF for HC Topology ($\kappa = 1$)

The decision function $CD/ERDETF$ is $erd(T) - \kappa.est(T, p(t))$. In fact, the performance of $CD/ERDETF$ varies according to the κ value. At high parallelism degrees ($\beta \geq 4$), the best performance of this heuristic is achieved at high enough κ value. Further, at very low β values ($\beta \leq 1$), the best performance of the heuristic is achieved at low κ value. However, at moderate parallelism degrees ($2 \leq \beta \leq 3$), the best performance of $CD/ERDETF$ depends to a larger extent on the α value, such that at low α values better performance can be achieved by using high κ value. Moreover, at high α values better performance can be achieved by using low κ value. This can be interpreted as achieving a balance between the two factors of the scheduling decision function. These are the approximation of task level ($erd(T)'s$) which indicate the priority of task T with respect to the whole computation graph, and the task earliest starting time ($est(T, p(T))$ which tries to reduce the processor idle time spent upon assigning T to $p(T)$.

At low values of α, the variance among the values of the communication edges is relatively low as the values of the communication edges are chosen randomly from a relatively small range. For example, at $\alpha = 0.5$, the value of the communication edge is limited between 2 and 8 units, while at $\alpha = 3$ the communication edge can take a random value between 11 and 49 units. This causes the communication edges of coarse grain graph problems to take comparable values, which in turns limit the ability of the $est(T)$ factor to effectively reduce the processor idle times. This is in addition to the little effect of the small communication requirements with respect

to the high task computation times in scheduling coarse grain computation graphs. This makes the $erd(T)$ part of the scheduling decision function, the dominating factor as the differences between the earliest starting times of ready tasks become negligible if compared to the deferences among the corresponding $erd's$ values. So multiplying the *est*'s by high κ bring more balance to the decision function, and makes the small differences in the earliest starting times of ready tasks more feasible.

This is in contrast to the situation at high values of α, where low κ value is required since the differences in the *est* values among ready tasks are high enough for the *est* factor to impose better utilization of the processors.

The above discussion can be extended to the HC topology. This is because changing the topology from FC to HC has the effect of increasing the communication requirements on the original graph, and this will only affect the used κ values. In fact, due to the inherited increase of the communication requirement over the HC topology, smaller κ values than those used in the FC case should be used accordingly. Empirical testing statistically showed that the appropriate value of κ is a function of β and $1/\alpha$ for each multiprocessor topology. Figures 5.11 and 5.12 show the required κ value for each parallelism degree β and granularity level α, for the FC and HC topologies respectively.

The number of iterations that $CD/ERDETF$ requires to discover its best performance, on the FC topology, varies according to the value of β and α, see figures 5.13 and 5.14. Generally one can notice that fixing β at a value and increasing

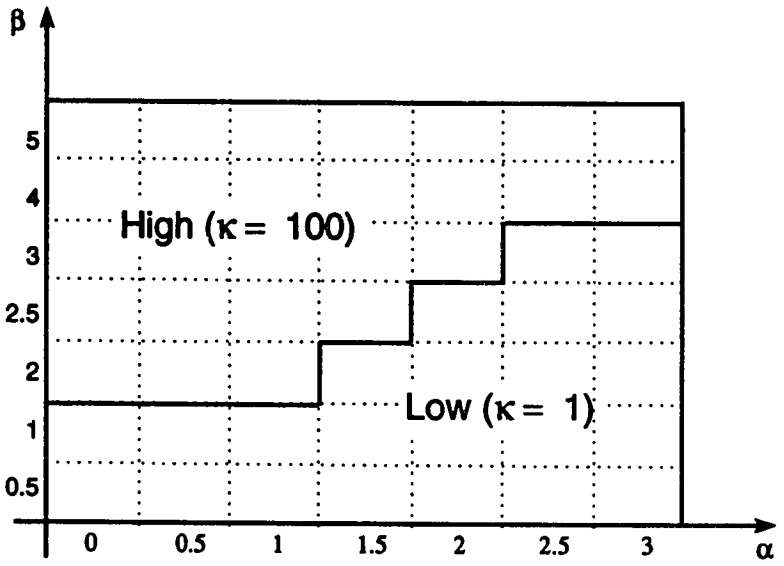

Figure 5.11: κ for the FC

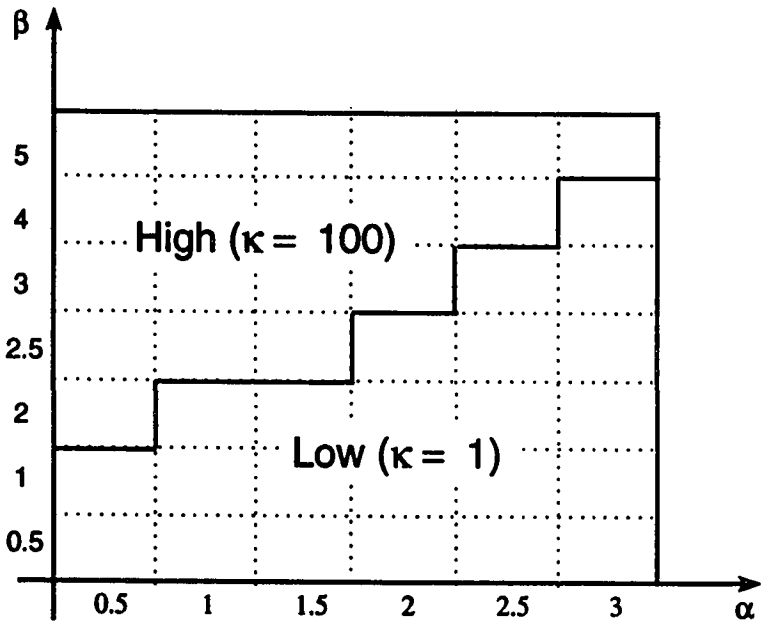

Figure 5.12: κ for the HC

the α, increases the required number of iterations. This increase in the number of iterations is due to the fact that increasing the communication requirement means increasing the average value of the communication edges of the *RGG* generated graph problems. As mentioned above, this in turns increases the variance among the communication edge values, which results in more irregularity in the computation graph that requires more iterations to investigate better solutions. For high parallelism degrees ($\beta \geq 4$), $CD/ERDETF$ with high κ value was able to find the best performance among all tested heuristics. with in the first 22 iterations. Further, at moderate parallelism ($2 \leq \beta \leq 3$) and low communication requirements ($0 \leq \alpha \leq 1$), $CD/ERDETF$ with high κ value was able to find the best performance among all tested heuristics with in the first 10 iterations.

The general behavior of $CD/ERDETF$ at low κ value, on the FC topology, with respect to the number of iterations required to find its best solution is that for the range $0.5 \leq \beta \leq 2$, the number of required iterations increases when β increase. However, for the range $2 \leq \beta \leq 5$ the number of required iterations decreases when β increases.

Again changing the topology from FC to HC has the effect of increasing the communication requirement on the original computation graph, but the general shape of the distribution is nearly maintained. See Figures 5.15 and 5.16.

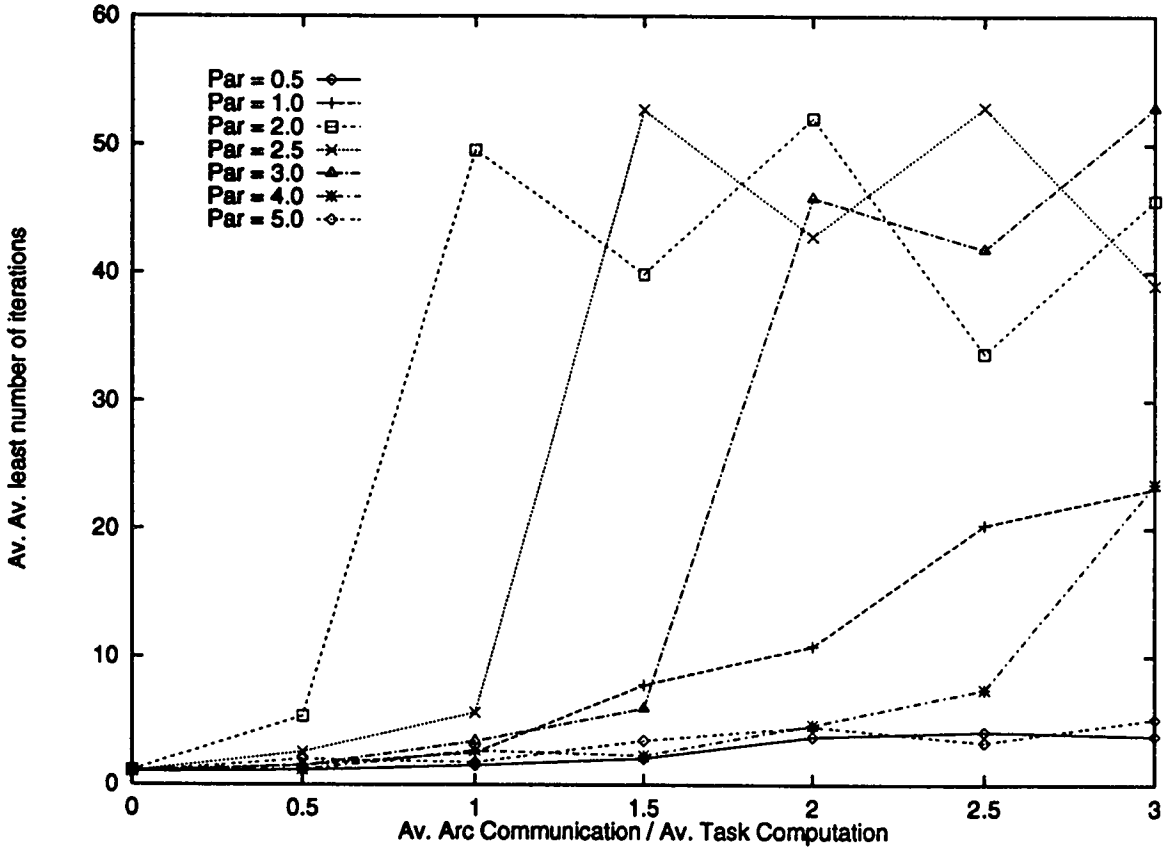

Figure 5.13: ALNI of CD/ERDETF for FC Topology ($\kappa = 100$)

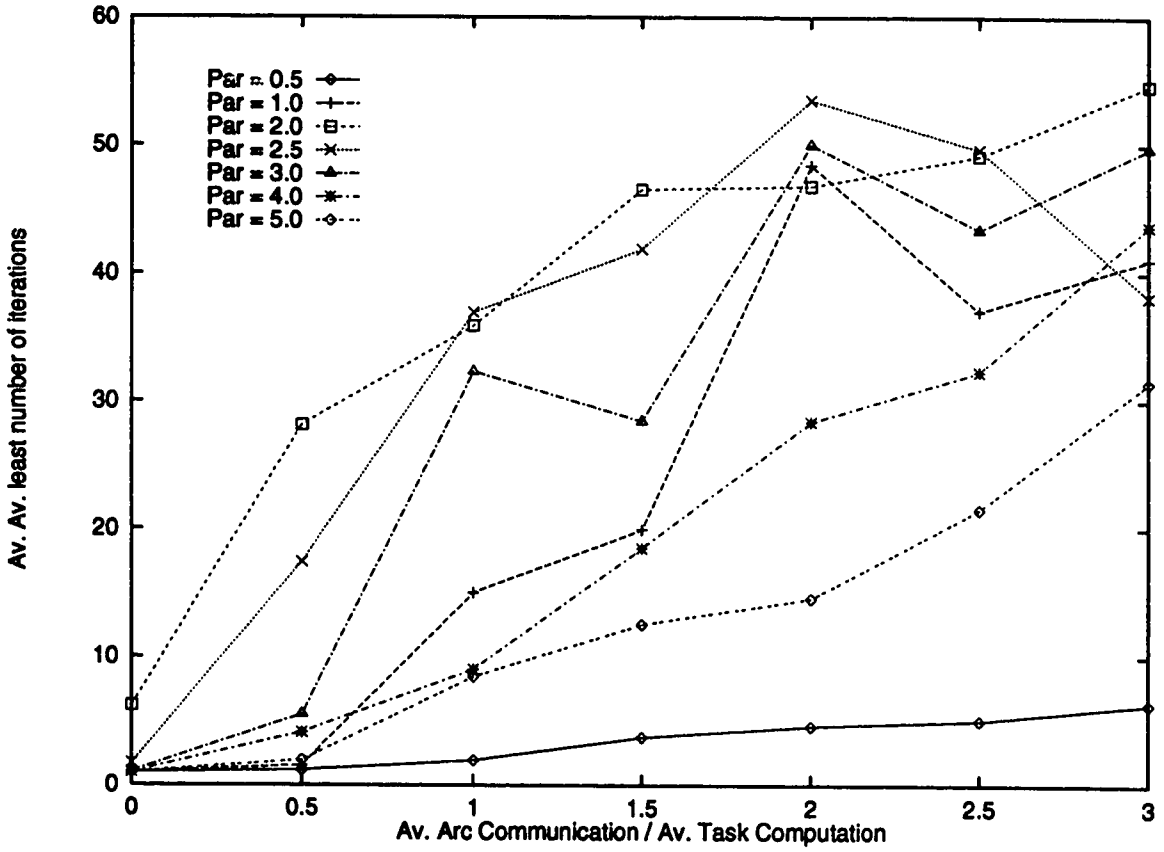

Figure 5.14: ALNI of CD/ERDETF for FC Topology ($\kappa = 1$)

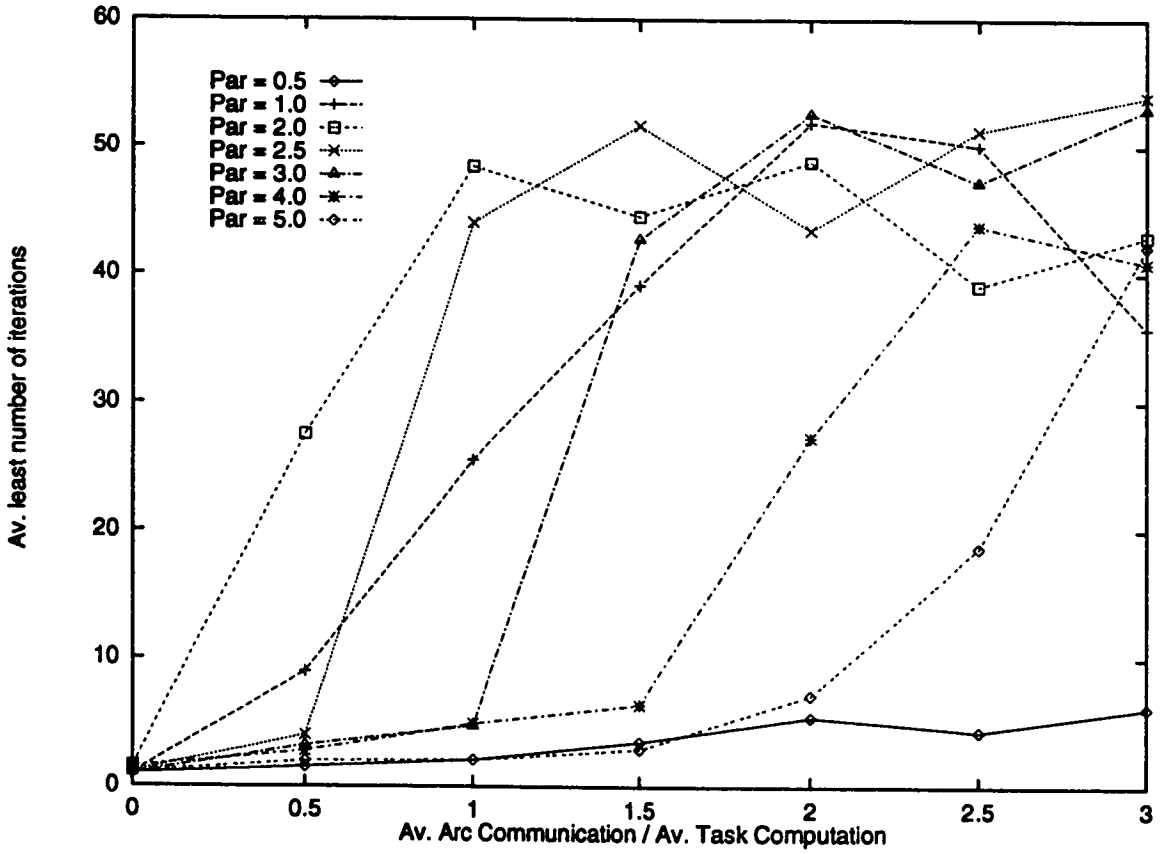

Figure 5.15: ALNI of CD/ERDETF for HC Topology ($\kappa = 100$)

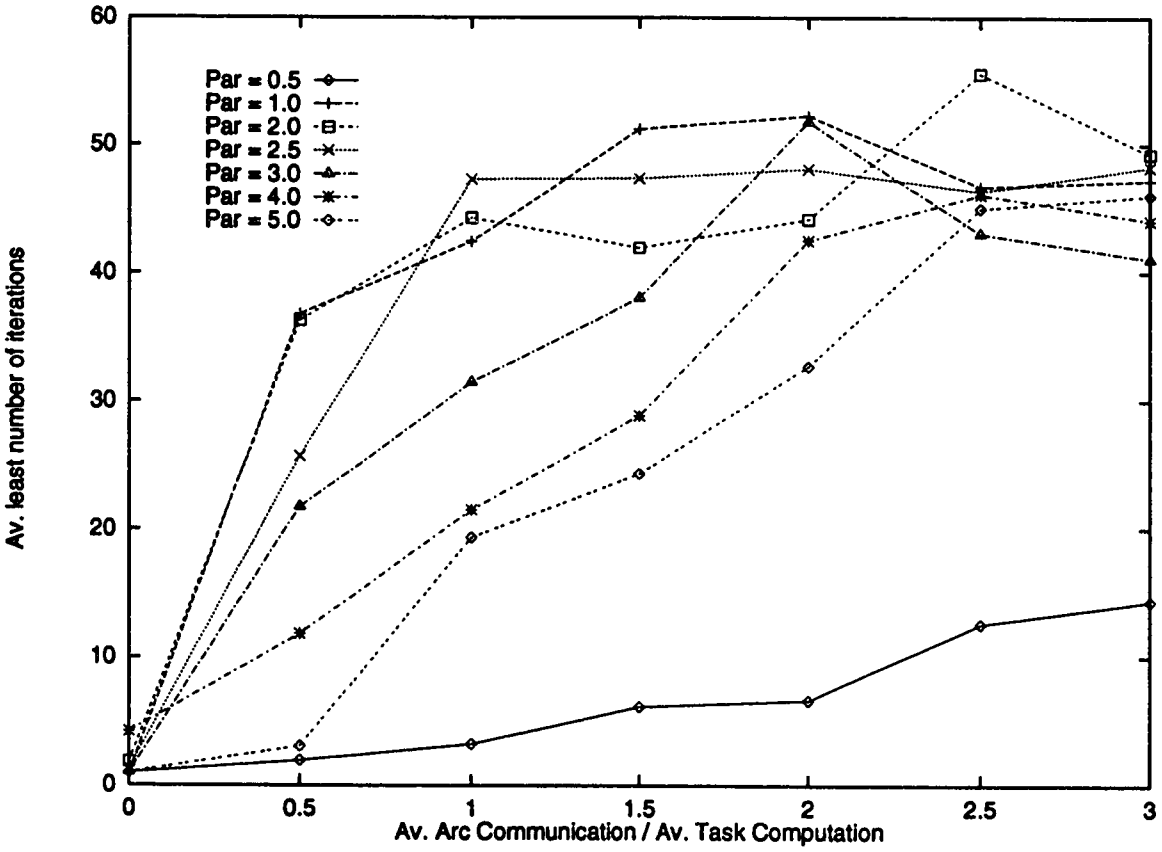

Figure 5.16: ALNI of CD/ERDETF for HC Topology ($\kappa = 1$)

5.5 Comparison to the Optimum Solution

Optimum random graph generator *ORGG* is another important graph-problem generator that can generate precedence computation graphs with various characteristics, where we know the finish times of the optimum schedules in advance. *ORGG* takes as its parameter an already randomly generated graph $G4$ along with its concurrency degree value and the communication to computation ratio, and uses a random scheduling heuristic to schedule G on the target machine. This heuristic will select randomly a task and assign it to the processor that can start it at the earliest. Then, *ORGG* fills the idle time intervals in the generated schedule with extra tasks, and assign for each filled task a computation time and if possible a predecessor task and a successor task. We call the new graph as modified random graph. *ORGG* tries its best to maintain the granularity level of the original input graph, as each filled task is given a computation time that does not break the task average computation time of the original graph, and the introduced communication edges to link this newly filled task to its predecessor and successor tasks are given values that do not violate the communication edge average value of the original graph. So, the communication to computation ratio of the original graph is maintained. By using the random heuristic, we aimed to reduce the chance of having any sort of graph-heuristic-dependence, among the generated optimum workload and the tested heuristics.

Using *ORGG* we have been able to generate random graph-problems for all pre-

viously studied values of α. When $\beta \geq 2.5$, $ORGG$ can not do appropriately in filling graphs at fine grain levels ($\alpha \geq 2.5$) and low parallelism ($\beta \leq 2$) specially on HC where the inherited communication requirement is higher. Due to this we considered the comparison for $\beta \geq 2.5$ on FC, and for $\beta \geq 3$ for HC. Figures 5.17 to 5.21 shows the performance of the three heuristics with respect to the finish times of the optimum schedules on FC topology. Also figures 5.25 to 5.27 shows the required number of iterations. We can notice that the general relative behavior of the heuristics is still maintained. At high parallelism $\beta \geq 4$, $CD/ERDETF$ with high κ value was able to produce the best performance, about (0.5%) from the optimum solution on FC, and for all studied levels of parallelism and communication requirements, Its peak deviation over FC is 1.5% away from the optimum solution. Similar conclusion can be extended for HC topology. However, $CD/HLETF^*$deviates by more than 5% at high parallelism on FC. Further, its peak deviation is 6.5% from the optimum solution for FC.

Our objective is achieved through the $CD/ERDETF$ scheduling heuristic that can maintain small finish time deviations over all studied problem instances, as the worst relative performance is 0.6% deviation on the FC topology and 0.8% deviation on the HC topology. Empirical testing proved statistically that $CD/ERDETF$ is just 0.5% by average away from the optimum solution over both the FC and HC topologies. Further, $CD/ERDETF$ is able to find its best performance with in much fewer number of iterations than that reqired by $CD/HLETF^*$.

5.6 Comparison to Other Scheduling Heuristics

Yong and Gerasoulais [34] used the PD/ETF heuristic as reference and compared their performance to that of Sarker [27]. Also we used PD/ETF in this work, therefore we can compare our work to that reported in [34]. They showed that their proposed strategy DSC [1] improved over the Sarker EZ significantly, slightly improved over PD/ETF. The best achieved performance of DSC over PD/ETF is 3.3% and this is at low granularity ($\alpha = 1$), while at higher α values the performance of DSC declined. Further, DSC requires additional processing to make the number of generated clusters meet the physical number of processors.

Reported performance comparison of seven scheduling heuristics [2] [22] using synthetic task graphs of various commonly encountered graph structures and parallel algorithms, showed that the dynamic critical pass DCP outperformed all the other heuristics. Pair wise comparison with PD/ETF showed that the performance DCP surpassed that of PD/ETF by up to 3.7 for completely random graphs, and the overall average best performance of DCP, for all studied graph structures and parallel algorithms, over that of PD/ETF is about 7%.

Our study indicates that the application of IRS on $CD/ERDETF$ significantly outperforms PD/ETF versus change in communication, parallelism, and network topology, and thus expected to outperforms these heuristics. Tables 5.1 to

[1] See section (3.3.1).
[2] DCP, PD/ETF, MD, MCP, DSC, EZ, and DLS.

5.8 indicate typical average percent deviations of PD/ETF, $CD/HLETF^*$, and $CD/ERDETF$ for all studied 4200 graph problems. In fact, $CD/ERDETF$ outperformed PD/ETF by up to 16%. So, we can say that the use of the iterative refinement with $CD/ERDETF$ significantly outperforms all the above scheduling specially for problem instances (fine-grain) where it is hard to find good solutions.

The cost of IRS is linear with the cost of $CD/ERDETF$ and can be implemented as a compiler optimization for programming distributed memory systems. It is specially useful for large-scale programs that are compiled once but repeatedly executed over different data sets.

Finally, Table 5.9 compares the time complexities and the processor requirements of several reported heuristics along with our proposed scheduling Heuristic.

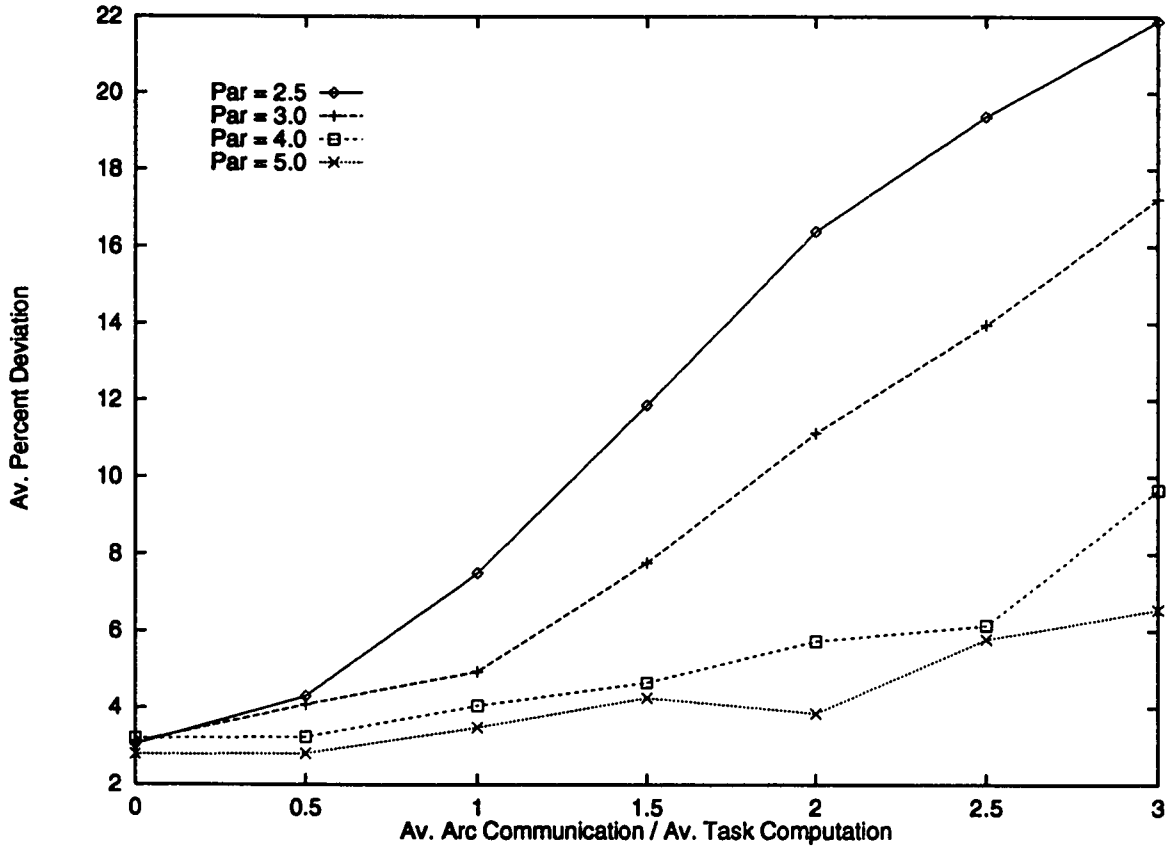

Figure 5.17: Performance of PD/ETF to optimum for FC Topology

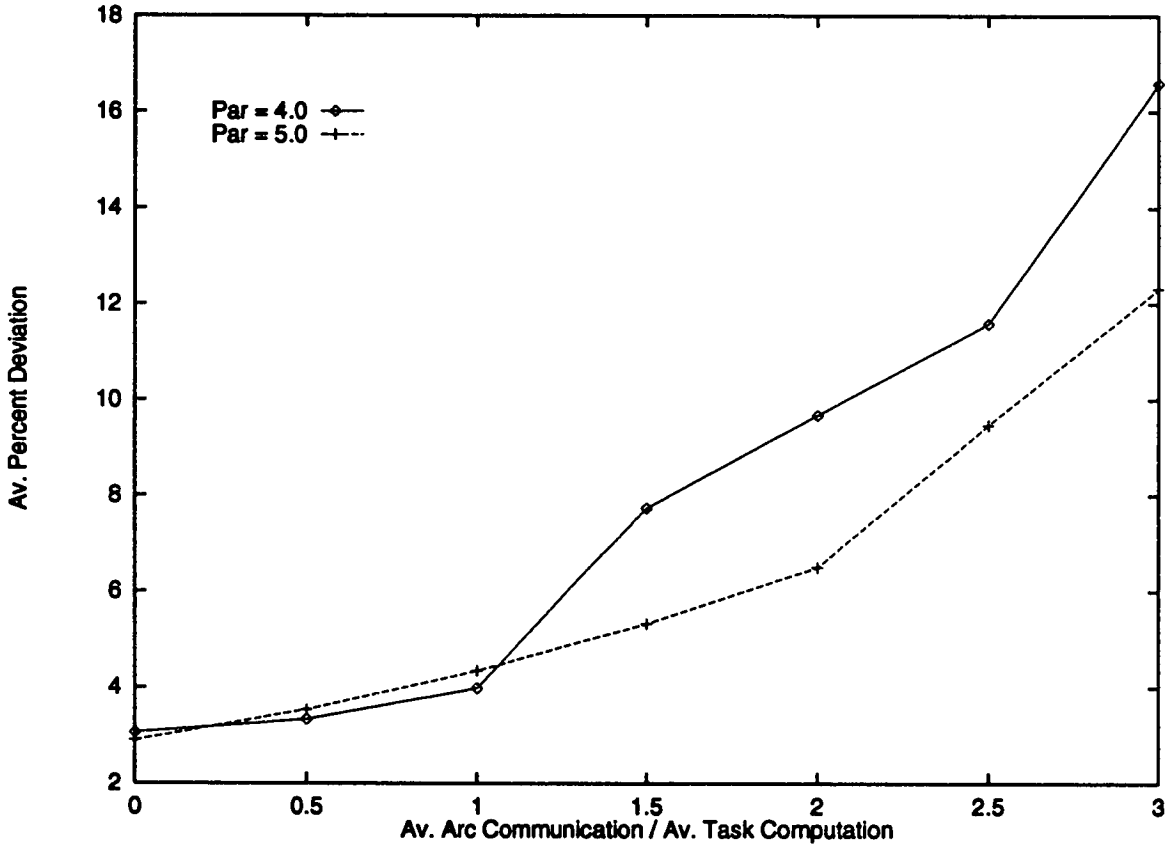

Figure 5.18: Performance of PD/ETF to optimum for HC Topology

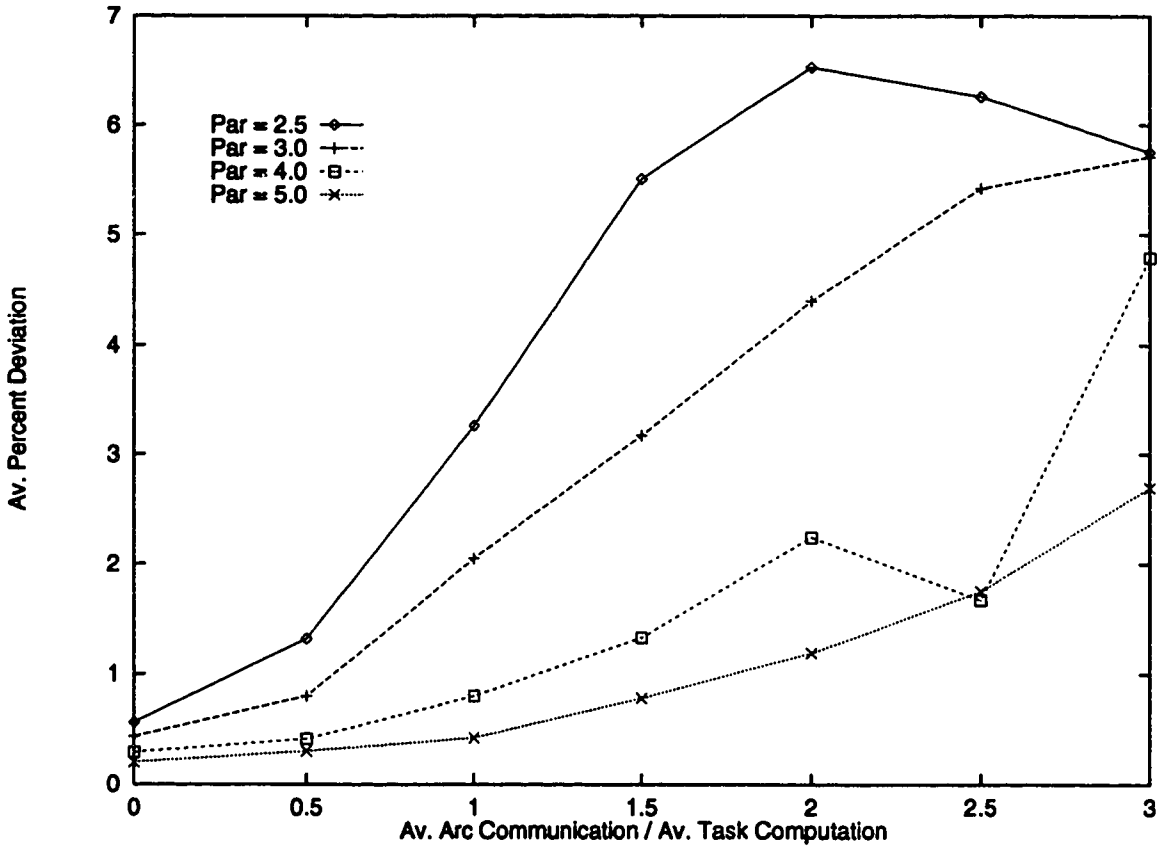

Figure 5.19: Performance of $CD/HLETF^*$ to optimum for FC Topology

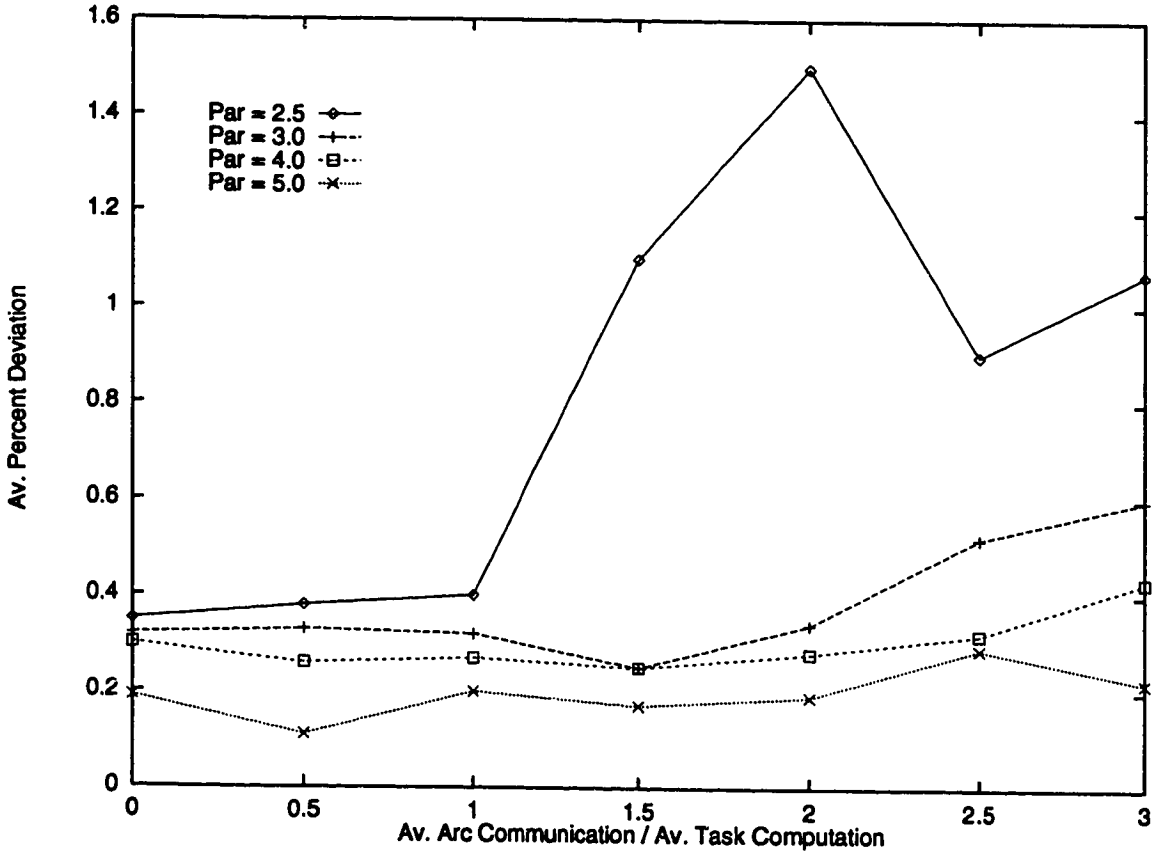

Figure 5.20: Performance of CD/ERDETF to optimum for FC Topology ($\kappa = 100$)

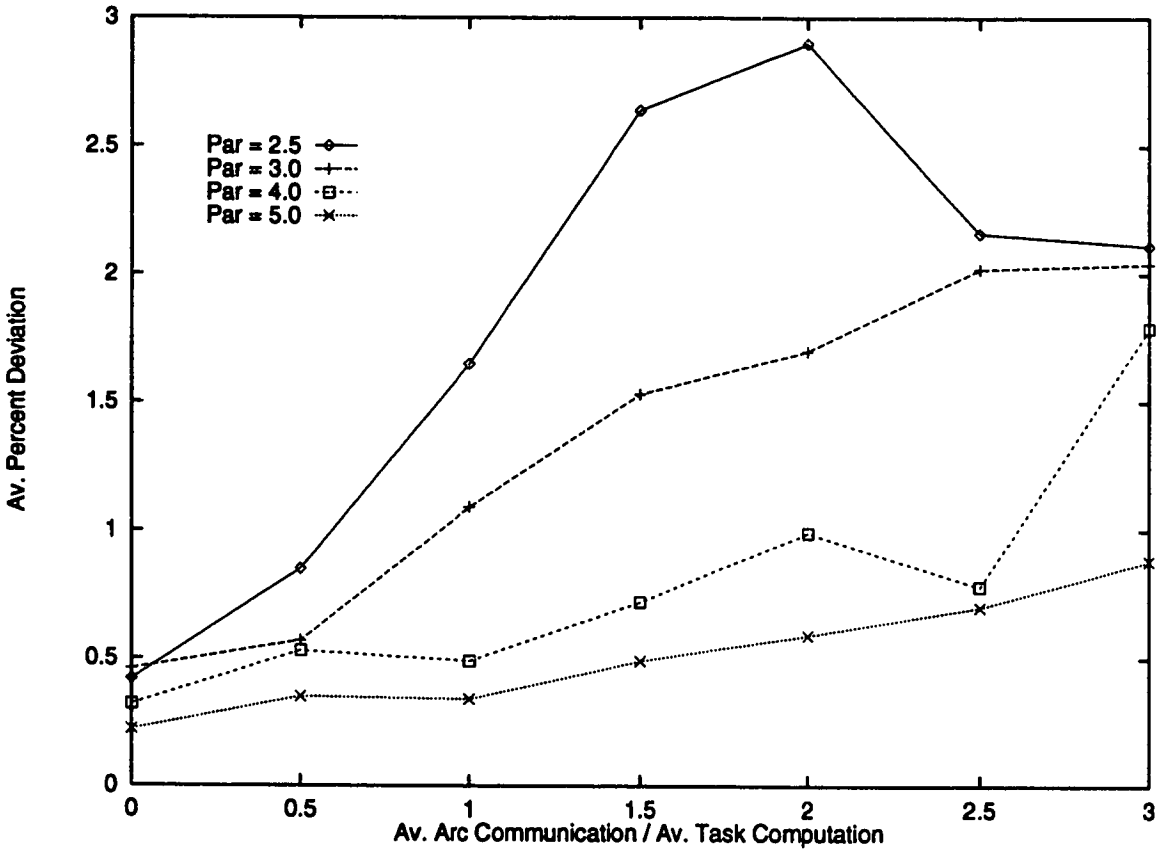

Figure 5.21: Performance of CD/ERDETF to optimum for FC Topology ($\kappa = 1$)

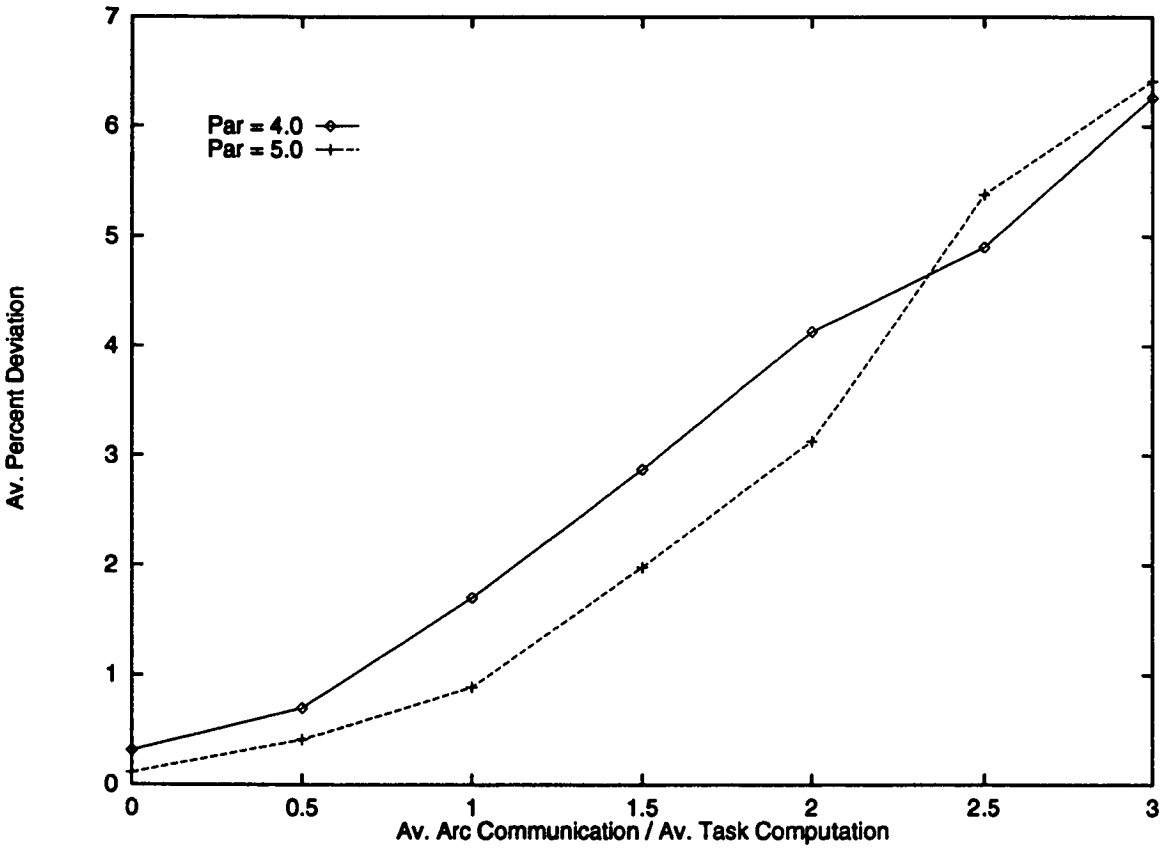

Figure 5.22: Performance of $CD/HLETF^*$ to optimum for HC Topology

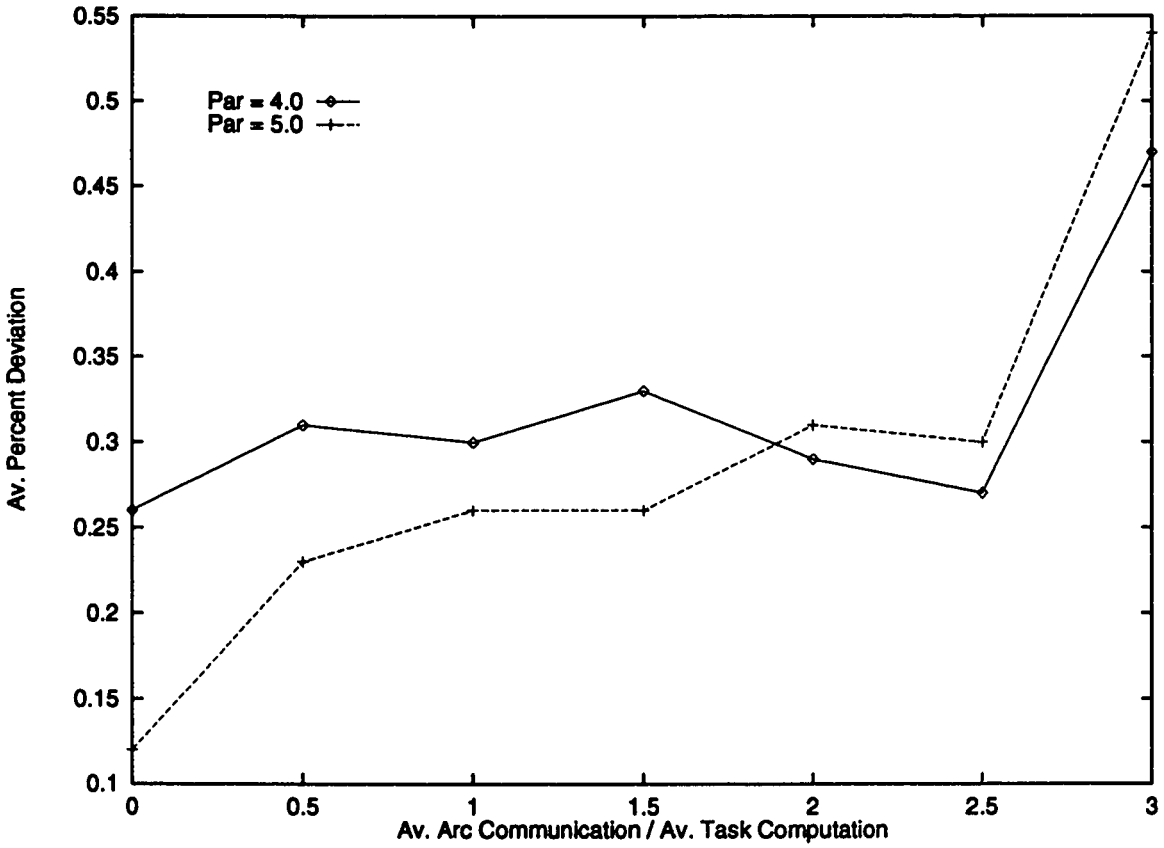

Figure 5.23: Performance of CD/ERDETF to optimum for HC Topology ($\kappa = 100$)

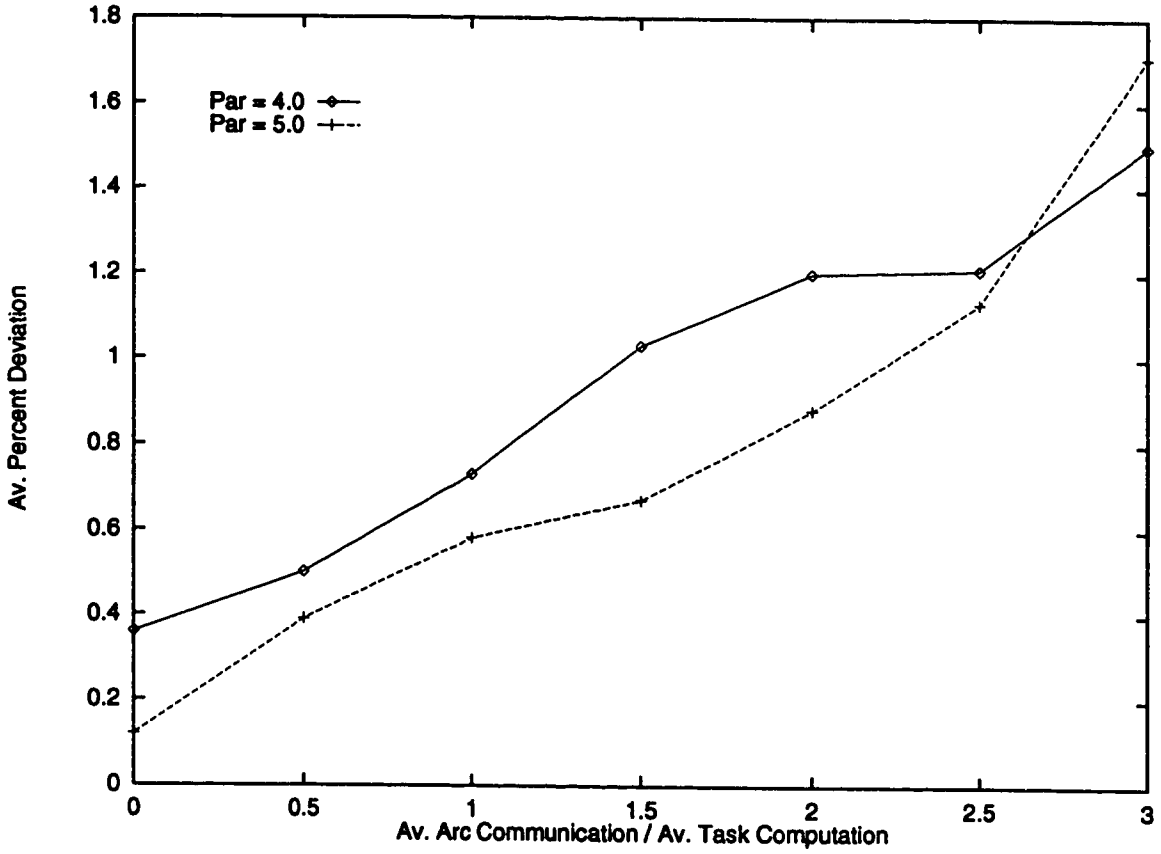

Figure 5.24: Performance of CD/ERDETF to optimum for HC Topology ($\kappa = 1$)

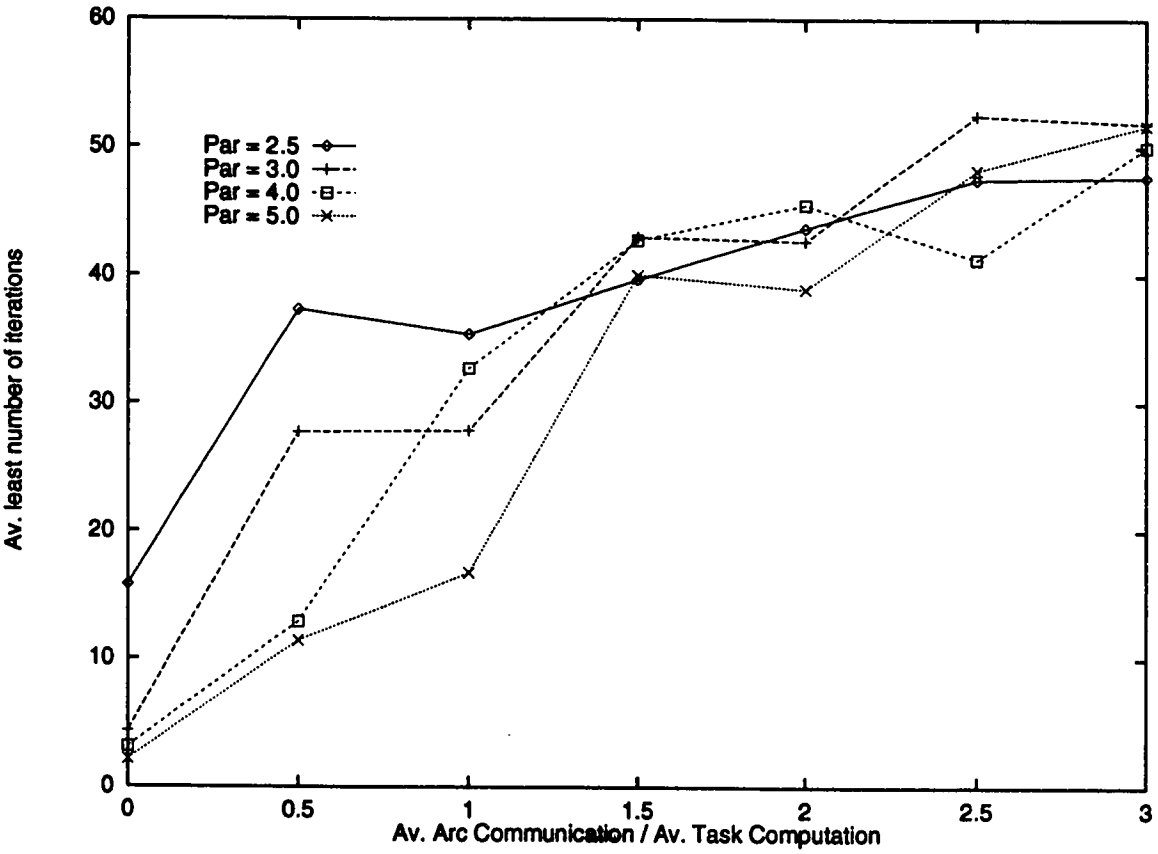

Figure 5.25: ALNI of $CD/HLETF^*$ for the optimum test on FC Topology

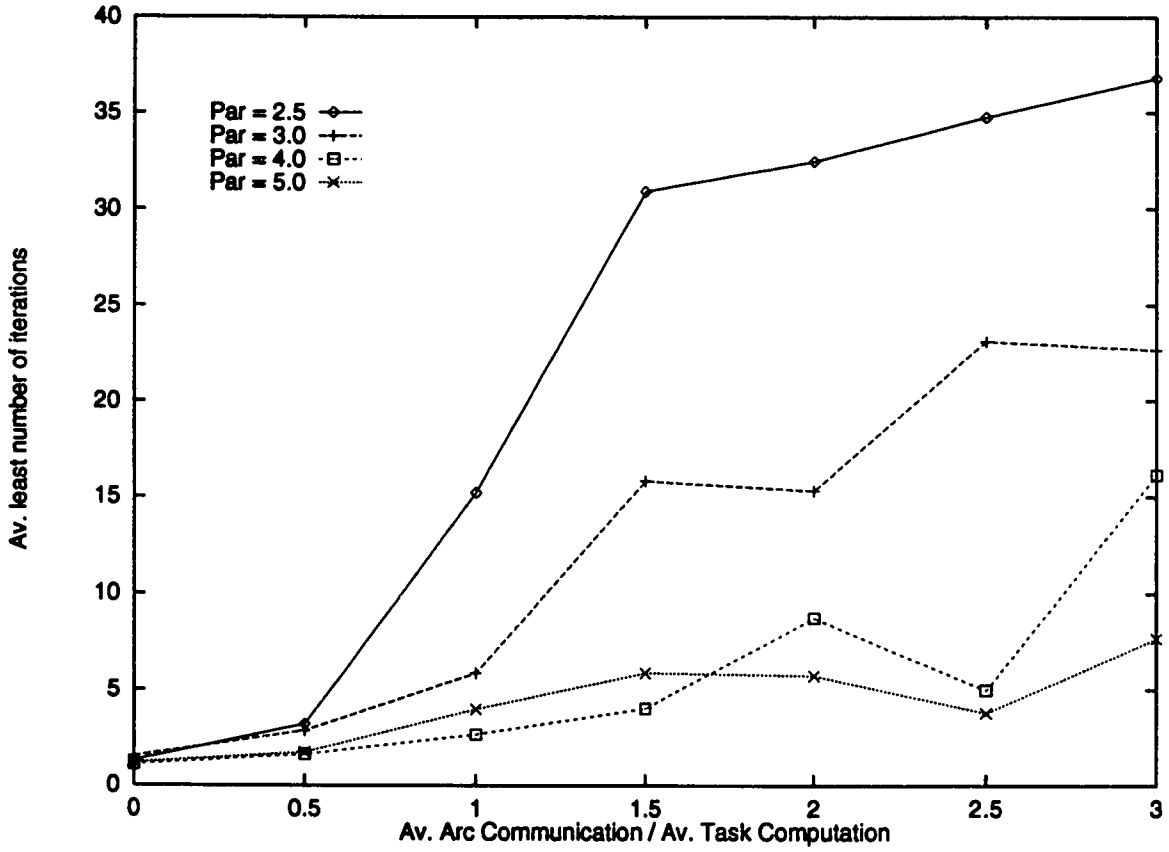

Figure 5.26: ALNI of CD/ERDETF for the optimum test on FC Topology ($\kappa = 100$)

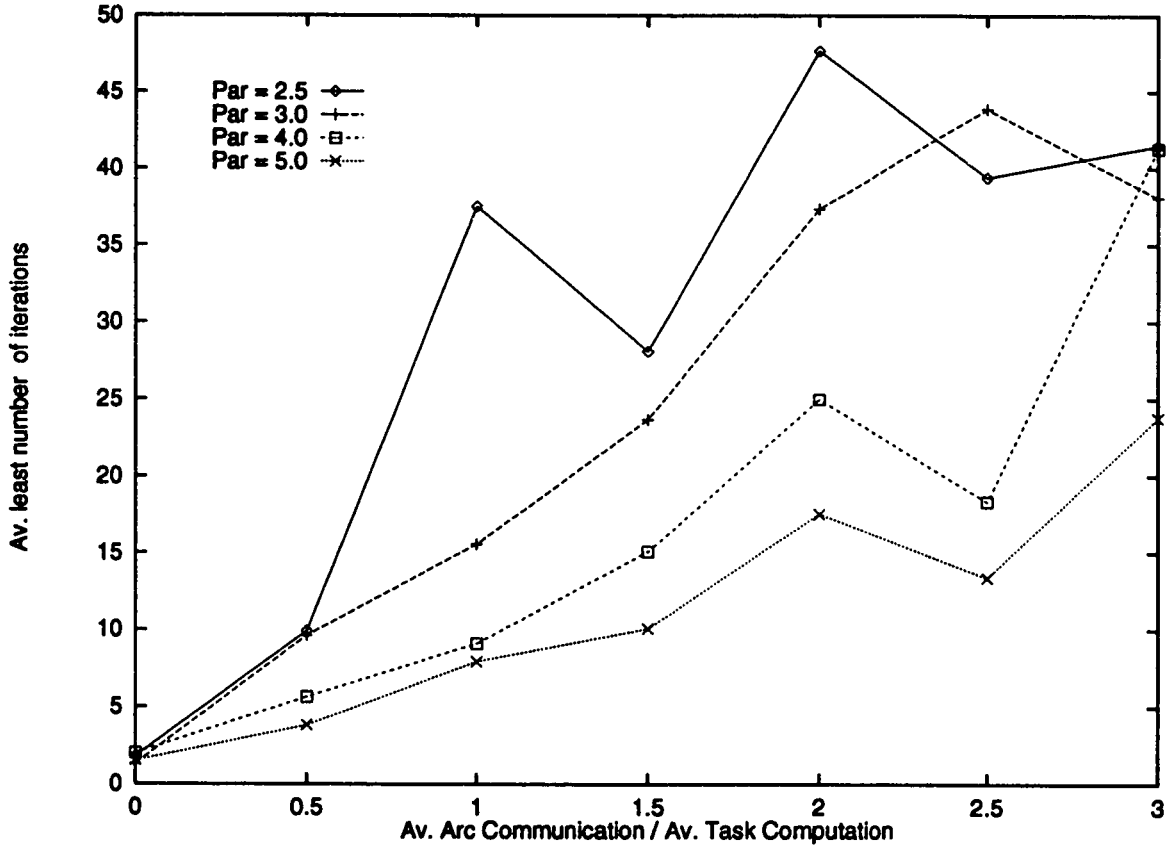

Figure 5.27: ALNI of CD/ERDETF for the optimum test on FC Topology ($\kappa = 1$)

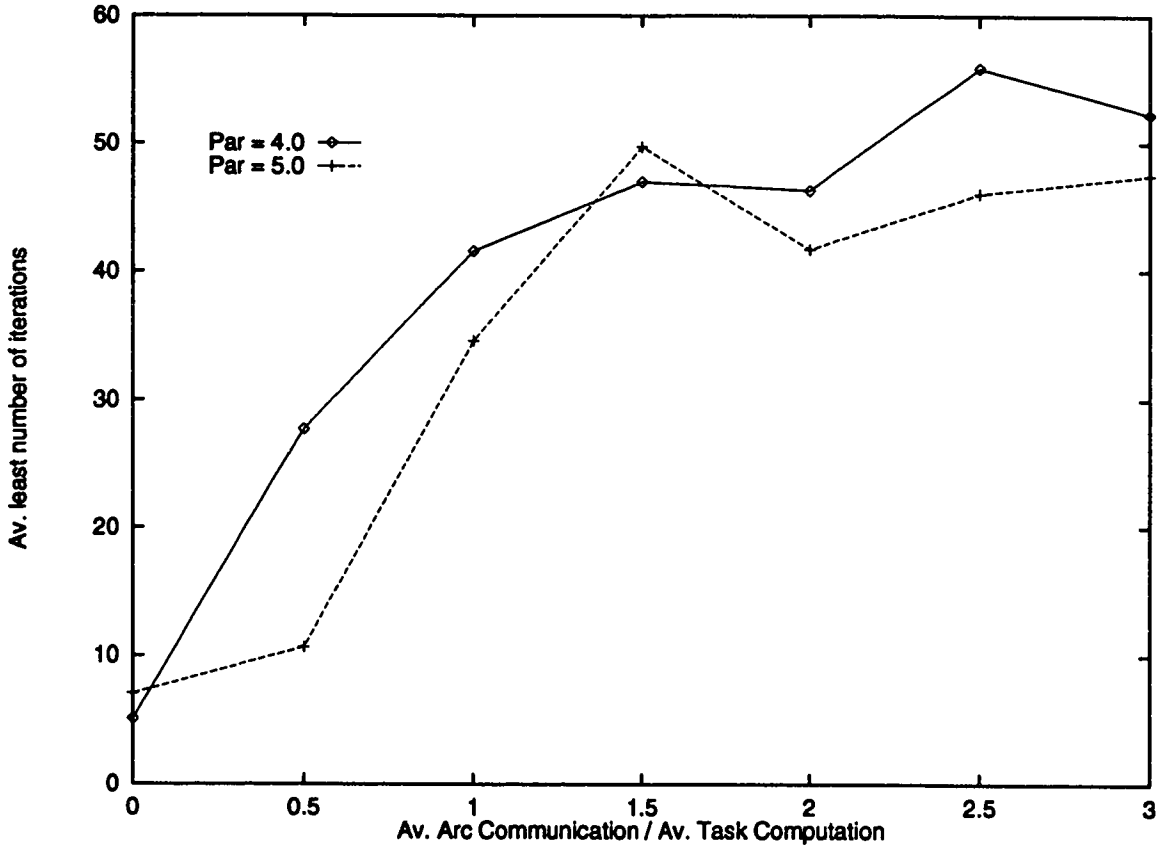

Figure 5.28: ALNI of $CD/HLETF^*$ for the optimum test on HC Topology

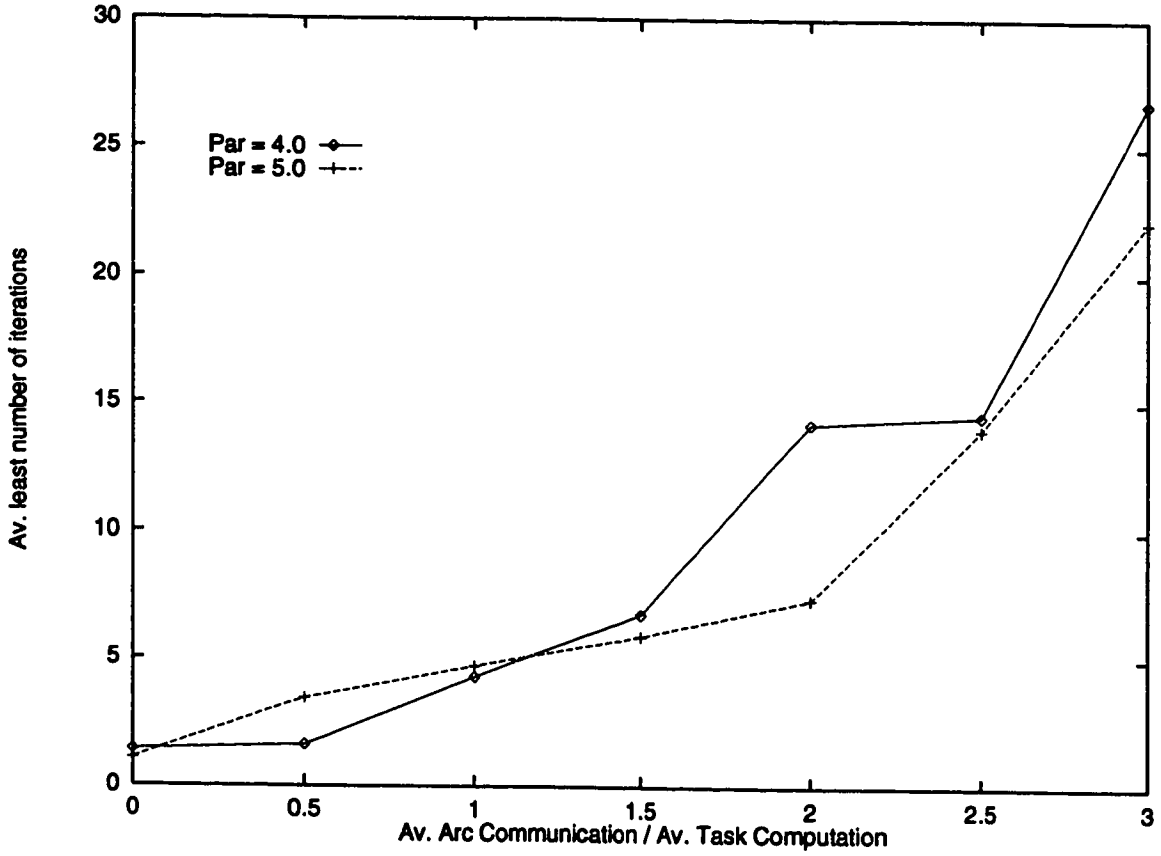

Figure 5.29: ALNI of CD/ERDETF for the optimum test on HC Topology ($\kappa = 100$)

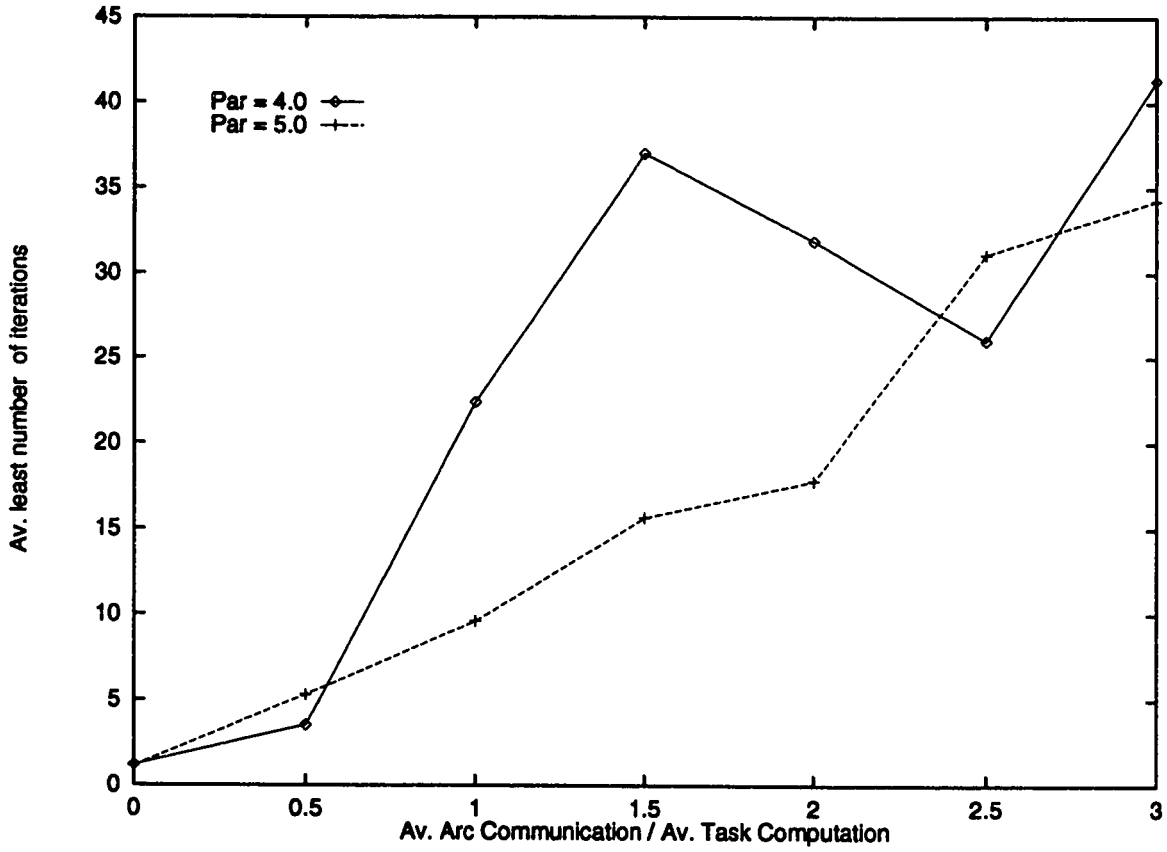

Figure 5.30: ALNI of CD/ERDETF for the optimum test on HC Topology ($\kappa = 1$)

Heuristic	$\beta = 1$		$\beta = 2$		$\beta = 3$		$\beta = 4$		$\beta = 5$	
α	low	high	low	high	low	high	low	high	low	high
PD/ETF	9.9	14.6	10.4	14.2	4	13.9	2.4	8.1	2.4	4.3
HLETF	0.6	1.1	1.9	1.4	1.3	3.3	0.4	3.2	0.2	1.2
$ERDETF_{(1)}$	0.2	0.6	0.9	0.6	1.1	1.3	0.5	1.8	0.3	0.6
$ERDETF_{(100)}$	1.8	1.6	0.5	0.8	0	0.3	0	0	0	0
$ERDETF_{(best)}$	0.2	0.6	0.5	0.6	0	0.3	0	0	0	0

Table 5.1: Comparison of the relative performance of *PD/ETF*, *CD/HLETF**, and *CD/ERDETF* with respect w_{best} on *FC* topology

Heuristic	$\beta = 1$		$\beta = 2$		$\beta = 3$		$\beta = 4$		$\beta = 5$	
α	low	high	low	high	low	high	low	high	low	high
PD/ETF	13.1	16.4	13.1	15.7	8	14.9	3.5	14.9	2.6	10.9
HLETF	0.5	0.8	2	1	2.3	1.9	1	4	0.4	3.7
$ERDETF_{(1)}$	0.4	0.9	0.8	0.7	1.4	0.5	0.8	1.4	0.3	1.8
$ERDETF_{(100)}$	2.1	1.1	0.9	1.3	0.1	0.9	0	0.7	0	0
$ERDETF_{(best)}$	0.4	0.9	0.8	0.7	0.1	0.5	0	0.7	0	0

Table 5.2: Comparison of the relative performance of *PD/ETF*, *CD/HLETF**, and *CD/ERDETF* with respect w_{best} on *HC* topology

Heuristic	$\beta = 1$		$\beta = 2$		$\beta = 3$		$\beta = 4$		$\beta = 5$	
α	low	high	low	high	low	high	low	high	low	high
HLETF	11.9	39.4	41.2	9.5	33.2	42.1	24	40.4	9.3	34.5
$ERDETF_{(1)}$	12.1	42.2	36.8	50.2	22.1	47.7	10.5	34.8	7.5	22.5
$ERDETF_{(100)}$	3.9	18.1	31.6	43.8	3.6	46.8	2.0	11.8	2.4	4.3
$ERDETF_{(best)}$	12.1	42.2	31.6	50.2	3.6	46.8	2.0	11.8	2.4	4.3

Table 5.3: Comparison of the *ALNI* of $CD/HLETF^*$, and $CD/ERDETF$ for w_{best} test on FC topology

Heuristic	$\beta = 1$		$\beta = 2$		$\beta = 3$		$\beta = 4$		$\beta = 5$	
α	low	high	low	high	low	high	low	high	low	high
HLETF	49.5	54.7	47	46.1	37.5	42.8	31.5	37.6	22.8	42
$ERDETF_{(1)}$	43.5	48.7	40.2	49.7	30.4	45.3	20.7	44.2	15.6	41.3
$ERDETF_{(100)}$	24.5	45.9	40.1	43.6	16.9	50.9	4.7	37.3	2.3	22.6
$ERDETF_{(best)}$	43.5	48.7	40.2	49.7	16.9	45.3	4.7	37.3	2.3	22.6

Table 5.4: Comparison of the *ALNI* of $CD/HLETF^*$, and $CD/ERDETF$ for w_{best} test on HC topology

Heuristic	$\beta = 2.5$		$\beta = 3$		$\beta = 4$		$\beta = 5$	
α	low	high	low	high	low	high	low	high
PD/ETF	7.9	19.2	5.6	14.1	4	7.2	3.5	5.4
$HLETF$	3.4	6.2	2.1	5.2	0.9	2.9	0.5	1.9
$ERDETF_{(1)}$	1.7	2.4	1.1	1.92	0.6	1.2	0.4	0.72
$ERDETF_{(100)}$	0.6	1.2	0.3	0.5	0.26	0.3	0.16	0.2
$ERDETF_{(best)}$	0.6	1.2	0.3	0.5	0.26	0.3	0.16	0.2

Table 5.5: Comparison of the performance of PD/ETF, $CD/HLETF^*$, and $CD/ERDETF$ with respect to optimum on FC topology

Heuristic	$\beta = 4$		$\beta = 5$	
α	low	high	low	high
PD/ETF	5	12.6	4.4	9.4
$HLETF$	1.8	5.1	3.3	5
$ERDETF_{(1)}$	0.8	1.3	0.6	1.2
$ERDETF_{(100)}$	0.3	0.3	0.3	0.4
$ERDETF_{(best)}$	0.3	0.3	0.3	0.4

Table 5.6: Comparison of the performance of PD/ETF, $CD/HLETF^*$, and $CD/ERDETF$ with respect to optimum on HC topology

Heuristic	$\beta = 2.5$		$\beta = 3$		$\beta = 4$		$\beta = 5$	
α	low	high	low	high	low	high	low	high
$HLETF$	37.5	46.3	32.8	49	29.4	45.6	22.7	46.3
$ERDETF_{(1)}$	25.2	42.9	16.3	39.8	9.9	28.2	7.3	18.2
$ERDETF_{(100)}$	16.4	34.7	8.2	20.4	2.7	14.5	3.8	5.7
$ERDETF_{(best)}$	16.4	34.7	8.2	20.4	2.7	14.5	3.8	5.7

Table 5.7: Comparison of the $ALNI$ of $CD/HLETF^*$, and $CD/ERDETF$ for the optimum test on FC topology

Heuristic	$\beta = 4$		$\beta = 5$	
α	low	high	low	high
$HLETF$	38.8	51.1	31.6	45.1
$ERDETF_{(1)}$	21	33.1	10.2	27.7
$ERDETF_{(100)}$	4.2	18.5	4.7	14.5
$ERDETF_{(best)}$	4.2	18.5	4.7	14.5

Table 5.8: Comparison of the $ALNI$ of $CD/HLETF^*$, and $CD/ERDETF$ for the optimum test on HC topology

Heuristic	$\sharp$ *processors*	*Time complexity*
Sarker (EZ)	Unbounded	$O(e(e+n))$
Yang and G. (DSC)	Unbounded	$O((n+e)logn)$
Gajski (MCP)	Bounded	$O(n^2logn)$
Gajski (MD)	Unbounded	$O(n^3)$
Hwang (ETF)	Bounded	$O(p.n^2)$
Al-Mass. ($CD/HLETF^*$	Bounded	$O(p.n^2)$
Ahmed (DCP)	Unbounded	$O(n^3)$
Kruatrachue (DSH)	Unbounded	$O(n^4)$
(CD/ERDETF)	Bounded	$O(p.n^2)$

Table 5.9: Comparison of processor requirements and time complexities

Chapter 6

Conclusion and Future Work

This chapter concludes our work with a brief summary of our attempt to reduce the number of iterations required by *IRS*. Suggestion for future research are also presented.

6.1 Conclusion

This thesis addressed the problem of scheduling precedence computations with communication costs on distributed memory systems. A set $\Gamma = (T_1, T_2, ..., T_m)$ of m computational tasks along with their computation times, precedence constrains, and inter-task communication costs, is represented by a directed acyclic graph. Given a parallel computation of m tasks and a distributed multiprocessor system of n identical processors, the scheduling problem can be defined as mapping the set of

computational tasks to the available processors so that the finish time of the resulting schedule is minimum while the precedence constrains are preserved.

The objective of this research is to find a more refined strategy that can be incorporated with in the Iterative Refinement Scheduling (IRS) concept, that was previously developed, to approximate the task level. The basic idea of IRS is simple and involves alternatively scheduling the given parallel graph in forward and backward passes, while passing the evaluated task levels from one scheduling pass to another. We have proposed a global scheduling heuristic ($CD/ERDETF$) that combines the refined estimate of task levels with management of processor idle times.

Evaluation of the proposed heuristic is conducted by altering the granularity level, parallelism degree, and system topology. Analysis showed that, at coarse-grain computations, better performance can be achieved by reducing processor idle times. For fine-grain computations, however, better performance requires higher precision selection of critical tasks. We statistically proved the superiority of $CD/ERDETF$ over recently reported heuristics, as it outperforms ETF and $CD/HLETF^*$ by up to 17% and 5%, respectively. Also, $CD/ERDETF$ generates solutions that are just 0.5% by average away from the optimum solution, under the above stated conditions. Further, the proposed heuristic maintains comparable performance over FC and HC topologies. This is expected as it uses a scheduling decision function that is topology dependent. The time complexity of the proposed heuristic is $O(pn^2)$, where p and n are the numbers of processors and tasks, respectively.

We used Pascal language to implement our software. Programs were compiled and run on *PC* Pentium-66 MHz machine running *DOS* – 6.2.

6.1.1 Reducing the number of iterations

We investigated reducing the number of iterations required by *IRS* to generate its best solution, for a given parallel graph of m tasks. Our attempt was motivated by learning from the previous history.

The iteration process is divided into two phases. The first phase is called the learning phase, and consists of the first ten iterations. In this phase, the best achieved forward schedule finish time is marked (ω_b). Also, we prepare a profile for each task that consists of the scheduling decision orders taken by each task. The number of the different decision orders that a task T took is denoted by $d(T)$. Further, we assign for each decision order taken by a given task a quality attribute that reflects how good this decision order has been for this task. So, the higher the quality value the better the corresponding decision order. The value of the quality attribute of a decision order d_1 that a task T took is denoted by $q(T, d_1)$, and is equal to the number of times T took this decision order, out of the ten forward iterations, divided by the sum of the corresponding schedule finish time deviations from $\omega_b - 1$. In this phase, the average number of decision orders is computed as

$$d_{av} = \sum_{T_i \in \Gamma} d(T_i)/m.$$

The second phase, is called the proposing phase, and consists of another five forward scheduling passes. In this phase we distinguished between a loose task, a don't care task, and a valid task. A task T is considered loose, if the number of taken decision orders $d(T)$ exceeds the average number of decision orders d_{av}. A task is considered a don't care task, if it took just one decision order in the learning phase. Further, a task is valid, if it is not a loose task nor a don't care task. Upon each forward scheduling pass, we propose for each valid task its best two decision orders, propose for each don't care task its decision order, and leave the set of loose tasks to be scheduled on line by the scheduling heuristic. Conflicts in the proposed decision orders among tasks are resolved by proposing a decision order to the more prior task first. After scheduling, we update the task profiles, and propose the new task decision orders by altering among the two best decision orders, of the set of valid tasks, according to the resulted schedule finish times.

In fact, the decision order of each task is resulted from a partial schedule , i.e. after scheduling all of its predecessors, and we found that combining task decision orders out of several schedules may not always reduce the number of required iterations.

In conclusion, we summarize our contributions as :

- Proposing a more accurate task level than that proposed by Al-Massarani and Al-Mohamed, which proved to be effective in sharpening the solution's finish time. Since the finish time is just 0.5%, by average, away from optimum.

- This work statistically proved that the choice of the weight κ in the scheduling decision function not only provide shorter finish times but also reduces the searching time through the iterative process.

6.2 Future Work

The following are some open problems for future research on static scheduling:

- This thesis assumes that the underlying communication subsystem is contention free. In practice, however, contention can occur when more than one processor want to send messages through common channels. An important goal for future research is to generalize this work to incorporate the contention effect.

- Empirical testing showed that the weight κ affects the performance of the proposed heuristic. In fact, we found that the value of κ , that produced the best performance, is proportional to the parallelism degree and inversely proportional to the granularity level. Studying the possibility of setting the value of κ, at each scheduling step, in a dynamic manner is really an important topic for future work.

- For future static scheduling research, this work motivates investigating a performance bound on the schedule finish times of the proposed heuristic.

- Also, this work motivates investigating more on the possibility of reducing the number of iterations required by the iterative process to produce it best performance.

Nomenclature

The following table summaries the notation used in this thesis.

Symbol	Meaning
$G(\Gamma, \rightarrow, \mu, C)$	**Task model: a precedence computations with inter-task communication costs**
$G(\Gamma, \rightarrow, \mu)$	**Task model: a precedence computations with no communication**
Γ	**Set of m computation tasks**
$\mu(T)$	**The computation (execution) time of task T**
$c(T_i, T)$	**Number of message unit to be sent from task T_i to task T**
CP	**A critical path of a given graph problem G**
$S(P, R)$	**System model: P represents processors, R routing latencies**

(Continued)	
$p(T)$	The processor where task T has been assigned
$r(p_1, p_2)$	The cost to route a message unit from processor p_1, to processor p_2
α	Average of communication-to-computation ratio
β	Average of parallelism degree
$pred(T)$	The set of immediate predecessor tasks of task T
$succ(T)$	The set of immediate successor tasks of task T
R	The set of ready to run tasks
lmt_i	Let T_i be a predecessor task of task T, then lmt_i is the last message time to T from its predecessor T_i
$lmt(T)$	The last message time to a given task T from all its predecessors
$est(T, p)$	The earliest starting time of task T on processor p
$est(T)$	The earliest starting time of task T on the most suited processor
$ct(T)$	The completion time of task T
$lap(T)$	The longest activity path of task T
$erd(T)$	The effective remaining distance of task T
$eft(P)$	The earliest free time of processor P

References

[1] T. L. Adem, K. chandy, and J. Dickson. A comparison of list scheduling for perrallel procesing systems. *Commm. ACM*, 17(12):685–690, Dec. 1974.

[2] I. Ahmed and Y. Kwork. On using task duplication in parallel program scheduling. under review with IEEE Transactions on Parallel and Distributed Systems.

[3] Adel M. A. Al-Massarani. *Priority-Based Scheduling and Evaluation of Precedence Graphs with Communication Times.* MS. thesis KFUPM, Computer Engineering Department, 1993.

[4] M. Al-Mouhamed. Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Transactions on Software Engineering*, 16(12):1390–1401, December 1990.

[5] M. Al-Mouhamed and A. Al-Massarani. Iterative scheduling of precedense constrained computations and communication on message-passing systems. under review with IEEE Transactions on Computers.

[6] Mayez Al-Mouhamed and Adel Al-Massarani. Performance evaluation of scheduling precedence constrained computations on message passing systems. *IEEE Transactions On Parallel Processing and Distributed*, 5(12):1317–1322, December 1994.

[7] Y. C. Chung and S. Ranka. Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors. *Proc. of Supercomputing*, pages 512–521, November 92.

[8] E. Coffman and R. Graham. Optimal scheduling for two-processor systems. *Acta Inform.*, 1:200–213, 1972.

[9] E. G. Coffman and etc. *Computer and Job-Shop Scheduling Theory.* John WIELY and SONS, 1976.

[10] H. El-rewini, T. G. Lewis, and H. Ali. *Task scheduling in parallel and distributed systems.* Prentice Hall, Inc., 1994.

[11] Hesham El-Rewini. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[12] Eduardo B. Fernandez and Bertram Bussell. Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Trasactions On Computers*, C-22(8):745–751, August 1973.

[13] M. J. Flynn. Very high-speed computing systems. *Procc.IEEE*, 54:1901–1909, 1966.

[14] M. R. Gary and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completness.* W. H. Freeman and Company, 1979.

[15] R. L. Graham. Bounds on multiprocessing timing anomolies. *SIAM J. Appl. Math.*, 17(2):416–429, March 1969.

[16] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, (5):287–326, 1979.

[17] T. C. HU. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, May 1961.

[18] J. J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Comput.*, 18(2):244–257, Apr. 1989.

[19] Kai Hwang and Faye A. Briggs. *Computer Arcitecture and Parallel Processing.* McGraw-Hill, 1984.

[20] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. on Comp.*, C-33(11):1023–1029, November 1984.

[21] B. Kruatrachue and T. G. Lewis. Grain size determination for parallel processing. *IEEE Trans. Soft. Eng.*, pages 23–32, January 1988.

[22] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical path scheduling: An effective technique for scheduling task graphs onto multiprocessors. IEEE Transactions on Parallel and Distributed Systems.

[23] E. A. LEE and J. C. Bier. Architectures for statically scheduled sataflow. *Journal of Parallel and Distributed Computing*, 10:333–348, 1990.

[24] E. L. Lwaler and D. E. Wood. Branch-and-bound methods: A survey. *Oper. Res.*, 14:699–719, July 1966.

[25] D. M. Pase. *A comparative analysis of static parallel scheduler where communication costs are significat.* PH.D. thesis, Oregon Graduate Ins. of Sci. and Technol., Oregon, July 1989.

[26] M. Prastin. *Precedence-Constrained Scheduling with Minimum Time and Communication.* MS. thesis, University of Illinois at Urbana-Champaign, 1987.

[27] V. Sarker. *Partitioning and Scheduling Parallel Progarms for Multiprocessors.* MIT Press , Cambridge, MA, 1989.

[28] R. Sethi. Scheduling graphs on two processors. *SIAM Journal of Computing*, 5(1):73–82, Mar. 1989.

[29] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection -constrained hetrogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, Feb. 1993.

[30] Abraham Silberschatz and James L. Peterson. *Operating System Concepts.* Addison Wesley, 1988.

[31] D. Towsley. Allocating programs containing branches and loops within a multiple processor system. *IEEE Transaction on Software Engineering*, SE-12(10), 1986.

[32] J. Uliman. Np-complete scheduling problems. *J, comput. System Sci.*, 10:384–393, 1975.

[33] MIN-YOU WU and Daniel D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[34] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, Sep. 1994.

Vita

- Homam Marwan Rashad Najjari
- Born in 1970 Aleppo, Syria
- Received the Bachelor degree in Computer Science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, 1993.
- Received the Master degree in Computer Science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, 1996